Tumbling Towers: Sprint Report 3

CS3217

Elvis Teo Chin Hao	A0218206M
Lee Yong Ler	A0219859J
Quan Teng Foong	A0223929X
Taufiq Bin Abdul Rahman	A0218081L

Requirements **7**
Overview 7
Features and Specifications 8
Core Gameplay 8
Additional Game Features 8
Navigation & Screens 9
Performance 10
Requirements 10
Launch Time 10
Runtime Metrics 10
User Manual 13
Menu Screen 13
Player Select Screen 14
Game Mode Selection Screen 15
Gameplay Screen 16
Pause Screen 17
Settings Screen 18
Achievements Screen 19
Game Win Screen 20
Game Lose Screen 21
Help Screen 22
How to Play 23
Horizontal Movement 23
Rotate Button 23
Downward Movement 24
Power Up Line 24
Power Ups 24
Score and Time 24
Pause Button 24
Game Modes 24
Survival 24
Sandbox 24
Race the Clock 24
Scale the Heights 25
Designs **26**
Top Level Organisation 26
Fiziks 27
Module Structure 27
FiziksEngine 28

FiziksBody 30
Runtime Structure 32
Receiving Physics Updates 32
Making Changes to FiziksBodies 32
Contact Begins/Ends 32
Design Considerations 33
FiziksBody 33
Interaction between the Model Object and FiziksBody 34
Further Discussion 39
FiziksShapeNode 39
FiziksBody 39
FiziksEngine 39
Game Engine 40
Module Structure 40
Design Considerations 41
God Class 41
Instantiation of AchievementSystem & StatsTrackingSystem 42
Game World 43
Module Structure 43
Runtime Structure 44
Rotating/Moving a block 44
Placing a block 44
SpecialPropertiesContactResolver 44
Design Considerations 45
SpecialPropertiesContactResolver 45
Event Manager 47
Module Structure 47
Runtime Structure 49
Registering & posting events 49
Design Considerations 50
Game Modes 54
Design Considerations 55
Game Mode Behaviour Classes 56
Creating a new Game Mode 59
How is this Possible? 60
Module Structure 62
Runtime Structure 63
Winning/Losing a game 63
Game Timer 65

Powerups 66
Module Structure 66
Runtime Structure 66
Obtaining a power up 66
Using a power up 67
Design Considerations 68
Deciding the next power up 69
Creation of Powerups 71
Activating power up 72
Carrying out effects of powerups 74
Informing View powerup availability 75
Adding a new powerup 75
Stats Tracking System 77
Module Structure 77
Runtime Structure 78
Design Considerations 78
Achievements System 80
Module Structure 80
Runtime Structure 81
Design Considerations 82
Storage 84
Module Structure 84
Design Considerations 84
GameUpdater 86
Module Structure 86
Runtime Structure 86
Game Loop - An example use of GameUpdater inside GameInstanceController 86
Design Considerations 87
InputSystem 88
Module Structure 88
Runtime Structure 89
Design Considerations 89
Swapping Input Systems 90
Customizing Input Systems 90
Adding More InputSystems 90
GameInstanceController 91
Module Structure 91
Runtime Structure 92
Game Loop 92

Multiplayer 93
Module Structure 93
Adding More Players 94
Design Considerations 96
Sound 97
Module Structure 97
Runtime Structure 98
Design Considerations 98
Singleton or Pass around? 98
Adding More Sounds 99
ViewAdapter 100
Module Structure 100
Runtime Structure 101
Design Considerations 102
View 103
Module Structure 103
Design Considerations 103
Reusability 103
View Modifiers 104
Settings 105
Module Structure 105
Runtime Structure 105
External libraries & third party modules 107
SwiftUI 107
NSNotificationCenter 107
SpriteKit 107
CADisplayLink 108
Core Motion 108
AVKit and Media Player 108
Testing 109
Strategy 109
Reflection 110
Evaluation 110
Sprint 3 110
Sprint 2 110
Sprint 1 110
Lessons 111
Sprint 3 111
Sprint 2 111

Sprint 1 111
Known bugs and limitations 111
Appendix **113**
Glossary 113
Test Cases 114
Integration tests 114
Detailed Task Schedule + Task List 120
Individual contribution and roles 123
GUI Sketches/Screenshots 125
Menu Screens 125
Game Screens 125
Game End 126
Other Screens 127

Requirements

Overview

Tumbling Towers is a singleplayer/multiplayer game where users aim to build a tower as high as possible using tetris blocks. The user has to aim to build a stable tower, as the tower is subject to physics and gravity, otherwise the tower will tumble. The user can play with 4 game modes, and with two different power ups. The game supports up to two players. The game is a recreation of the popular Nintendo Switch game, Tricky Towers.

In this application, we intend to demonstrate the clean modularisation of our application, i.e. the separation of the functionality of our application such that we are able to replace parts of the application easily. We also intend to show the extensibility of our application, i.e. the ease of addition of new features such as new power ups, new game modes, input modes, and more players.

In Sprint 1, we focused on setting up the game and physics engines (which represents the model of our game) rendering of the blocks using SwiftUI, and controlling the blocks using gyroscopic and touch input. Our aim was to get the core functionality of the application (falling tetris blocks which can be controlled) up and running.

In Sprint 2, we focused on adding features which would enhance our game, such as game modes, power ups, rotation of the blocks and the screens to allow navigability to the game. In addition, we did some refactoring of our game and physics engines.

In Sprint 3, we split the sprint into two sections (3.1 and 3.2).

In 3.1, we added the entire navigability (pause, player select, game mode select), as well as added local multiplayer (where >1 game screens would be displayed on the iPad). We also refactored the game engine in 3.1. The rendering of the achievements screen and storage of achievements, stats and settings was also added. 2 new game modes have also been added.

In 3.2, we refactored the View and View Model, to modularise the View by facading out SwiftUI, the current View framework that we are using. We have also cleaned up the code, wrote more test cases and worked on the documents.

Features and Specifications

Core Gameplay

These features define the core premise of the game, being able to build a tower using tetris blocks.

S/N	Feature Requirement
1.1	Blocks drop from above, and upon landing are subject to physics
1.2	When the input mode is set to be gyro, user can control the block's horizontal position while the block is falling by tilting the device
1.3	When the input mode is set to tap, user can control the block's horizontal position while the block is falling using their finger
1.4	The blocks that spawn should be of one of these types: - O block - I block - J block - L block - T block - Z block - S block
1.5	The user can speed up the descent of the block
1.6	The user can rotate the block
1.7	User should be able to pause the game, such that when the game resumes, the blocks are in the position they were just before pausing the game

Additional Game Features

These features are additional - they serve to enhance and improve the gaming experience of users by adding interesting elements and objectives to the game.

S/N	Feature Requirement
2.1	There should be 4 game modes with varying objectives: - Survival - Race the Clock - Scale the Heights - Sandbox

2.2	There should be 2 power ups: - Glue - Platform
2.3	The game should support singleplayer & multiplayer
2.4	The game should have background music and certain sounds should play upon certain events occurring: - Block being placed - Different sounds for activating different power ups - Power up being collected
2.5	There should be a score system that could calculate scores based on: - Time left - Number of blocks placed - Number of blocks dropped out of the level The way that score is calculated can be different for different game modes
2.6	There should be a visual column guide under the currently moving block showing the path that it is taking

Navigation & Screens

S/N	Feature Requirement
3.1	The menu screen should display navigation buttons to different screens
3.2	The settings screen should allow adjusting of game volume and input mode
3.3	The achievement screen should show achievements that have been attained
3.4	The player select screen should allow selecting of the number of players: - Singleplayer - Multiplayer (2 players)
3.5	The gameplay screen should allow players to play the game. It should have UI elements to display the score and the time.
3.6	The game mode screen should allow selection of game mode
3.7	The pause screen should allow the user to either continue or exit the game
3.8	The help screen should show information about the game to the user

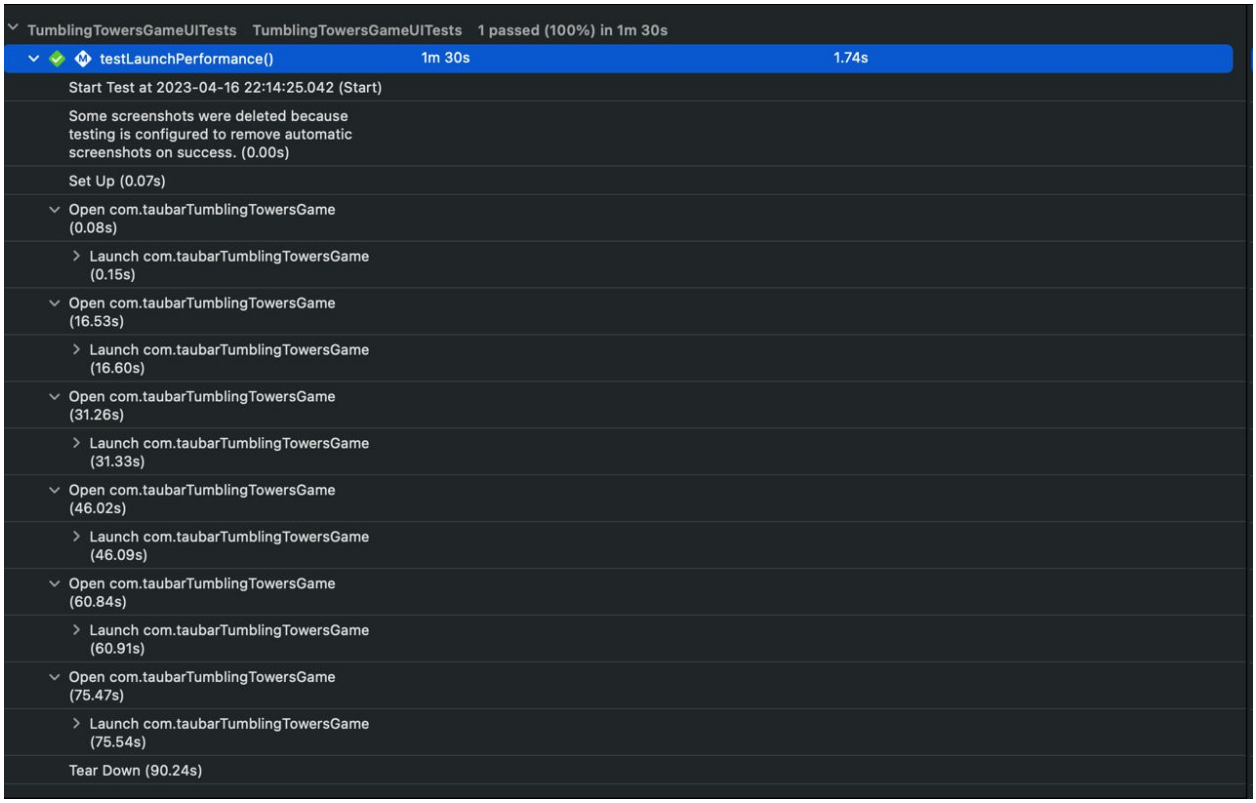
Performance

Requirements

The system should be an iPad running iOS 15.5 and above.

Launch Time

According to Apple, it is [recommended that app launch times do not exceed 400ms](). From our launch tests, we see that the launch times are all under 100ms:



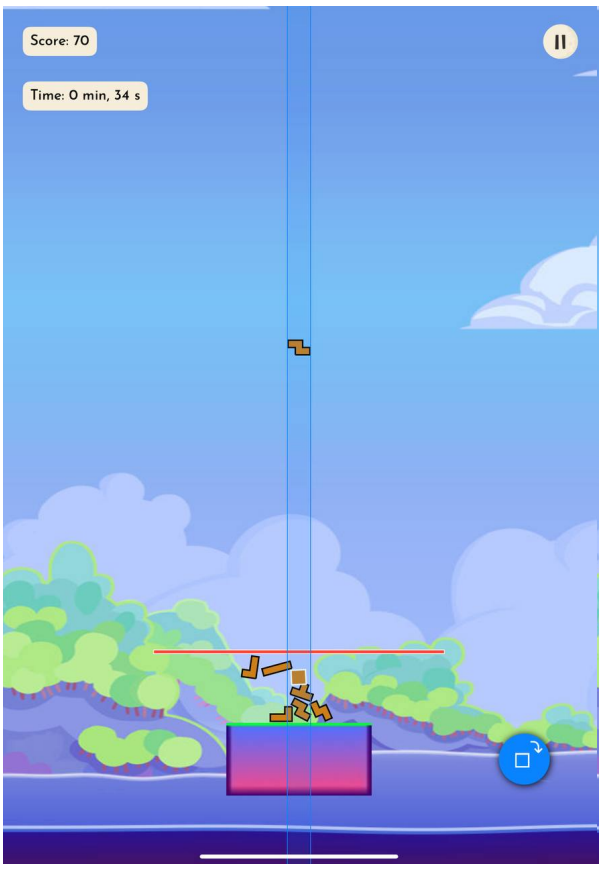
Runtime Metrics

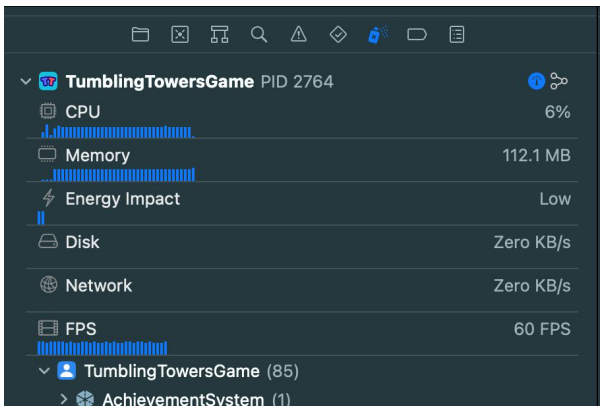
Below are two states of the game.

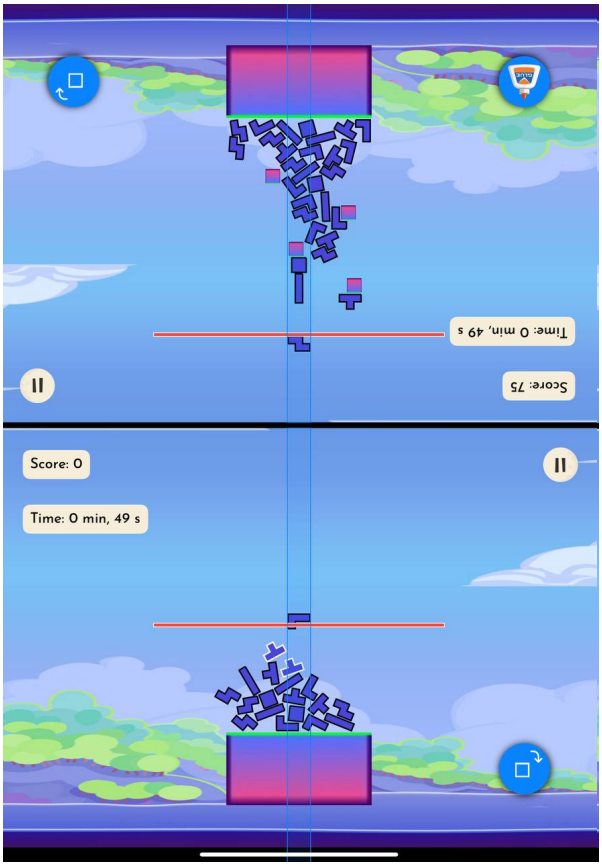
The first state is when the game has just begun (few blocks), and it is seen that the CPU usage is 6% and the memory usage is 112.1MB.

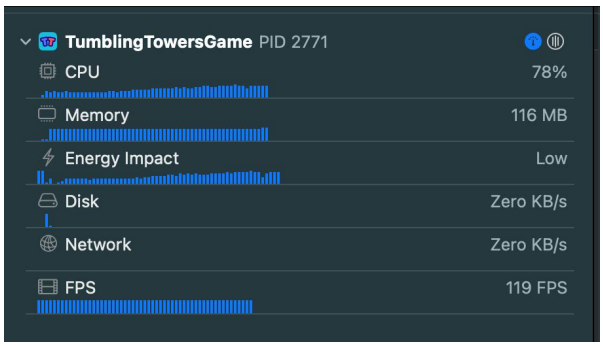
The second state is when a multiplayer game is nearly ending (many blocks and platforms), and both users are controlling at once. It is seen that the CPU usage is 78% and memory usage is 116MB.

These tests were run on an iPad Air.


Score: 70
Time: 0 min, 34 s


TumblingTowersGame PID 2764
CPU
6%
Memory
112.1 MB
Energy Impact
Low
Disk
Zero KB/s
Network
Zero KB/s
FPS
60 FPS
TumblingTowersGame (85)
AchievementSystem (1)

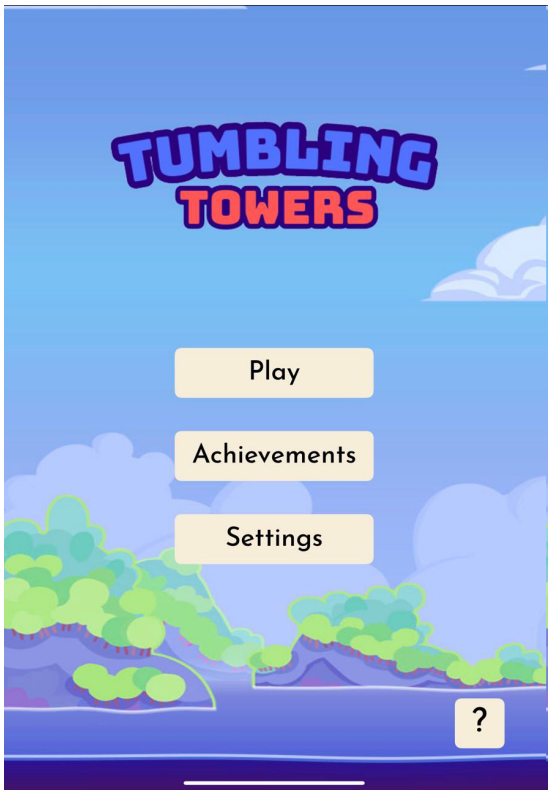

Score: 0
Time: 0 min, 49 s
Score: 75
Time: 0 min, 49 s


TumblingTowersGame PID 2771
CPU 78%
Memory 116 MB
Energy Impact Low
Disk Zero KB/s
Network Zero KB/s
FPS 119 FPS

User Manual

Menu Screen

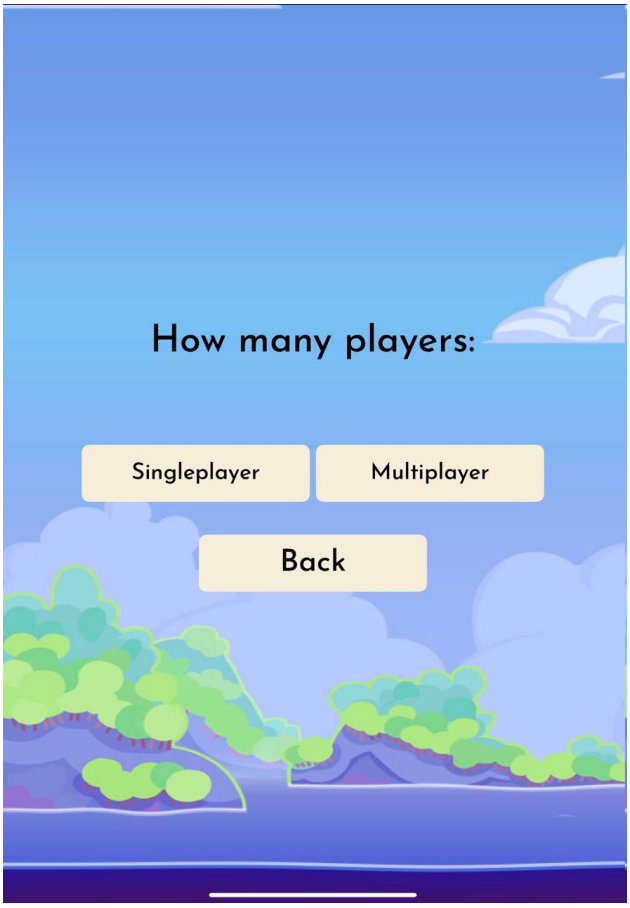
Upon opening the game, the user will see the Menu screen:



Menu Screen

Player Select Screen

Upon pressing the “Play” button on the Menu Screen, the user will be directed to the Player Select screen, where user can select how many players are playing:



Player Select Screen

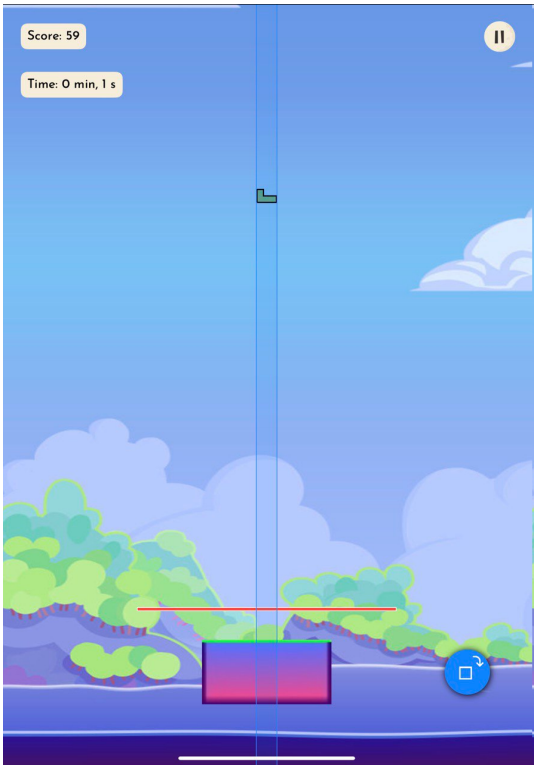
Game Mode Selection Screen

Upon choosing the number of players, the user will be directed to the Game Mode selection screen. Here, the user(s) can select the game mode that they wish to play:

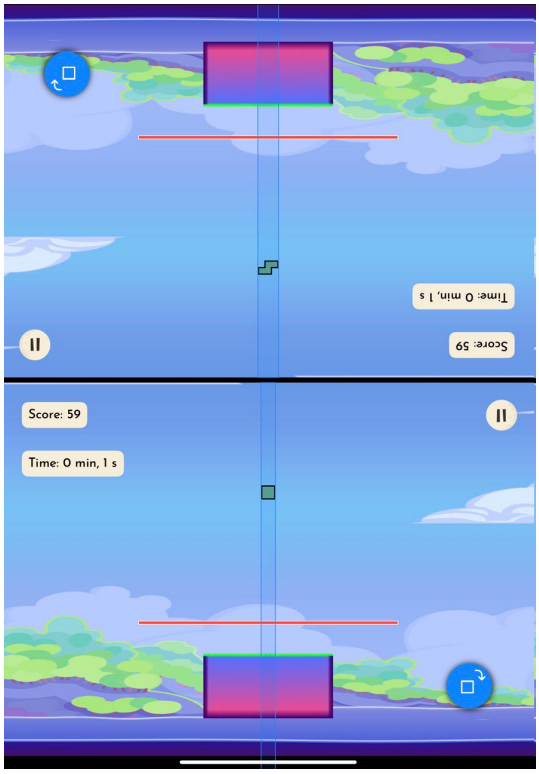
Game Mode Selection Screen

Gameplay Screen

After selecting the game mode, the game will begin. Depending on the number of players selected in the Player Select Screen, it can look like one of the following:



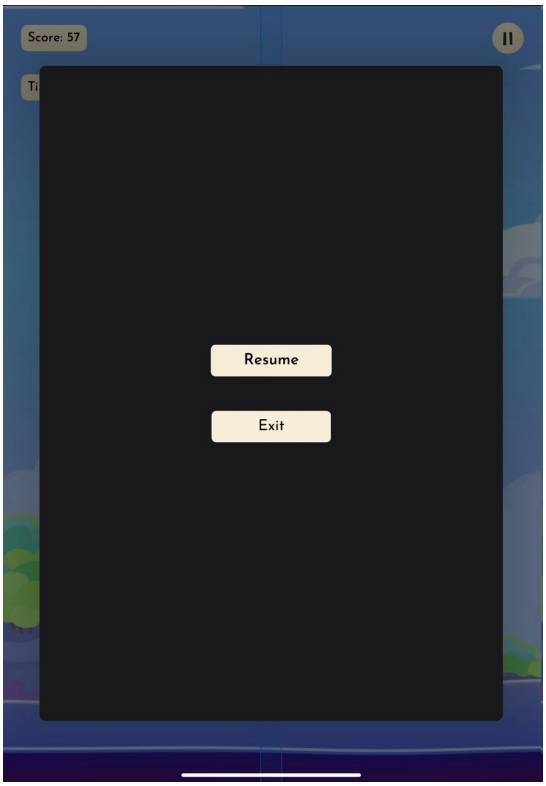
Gameplay Screen (Singleplayer)



Gameplay Screen (2 players)

Pause Screen

When user taps the pause button at the top right of the Gameplay screen, the pause screen will appear:

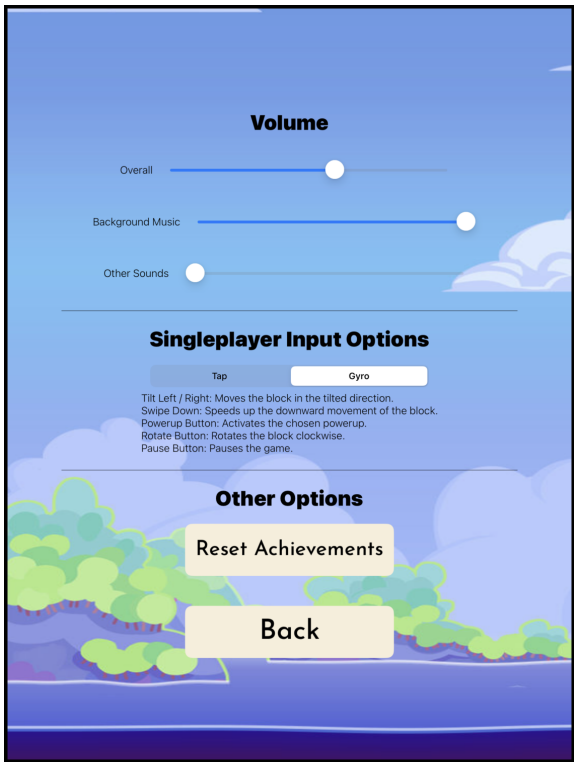


Pause Screen

The user can either resume the game by tapping the “Resume” button or exit the game by tapping the “Exit” button.

Settings Screen

If user taps the “Settings” button on the Menu screen, they will be directed to the Settings screen:

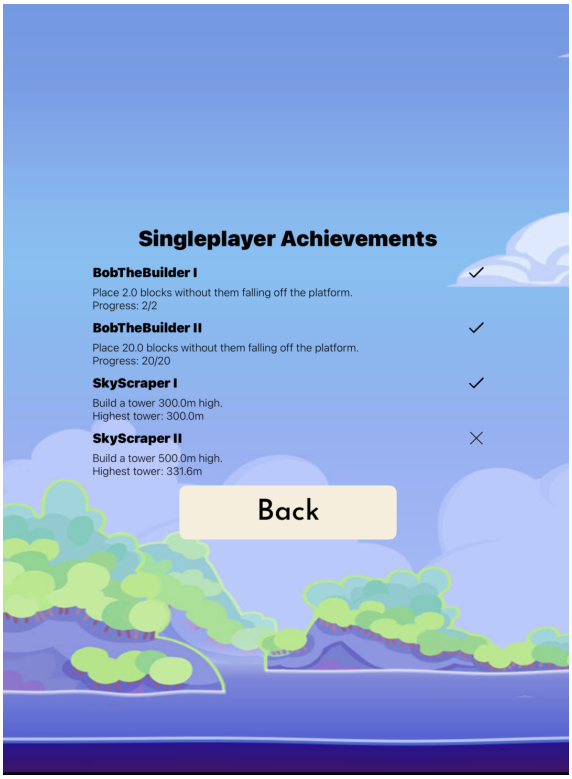


Settings Screen

The user can adjust the different volumes of the game (Overall, Background Music, Other Sounds) as well as change their Singleplay input mode option (Gyro or Tap). The user’s achievement’s progress could also be reset using the “Reset Achievements” button. The settings will persist to the next game session.

Achievements Screen

If the user taps the “Achievements” button on the Main screen, they will be directed to the Achievements Screen:

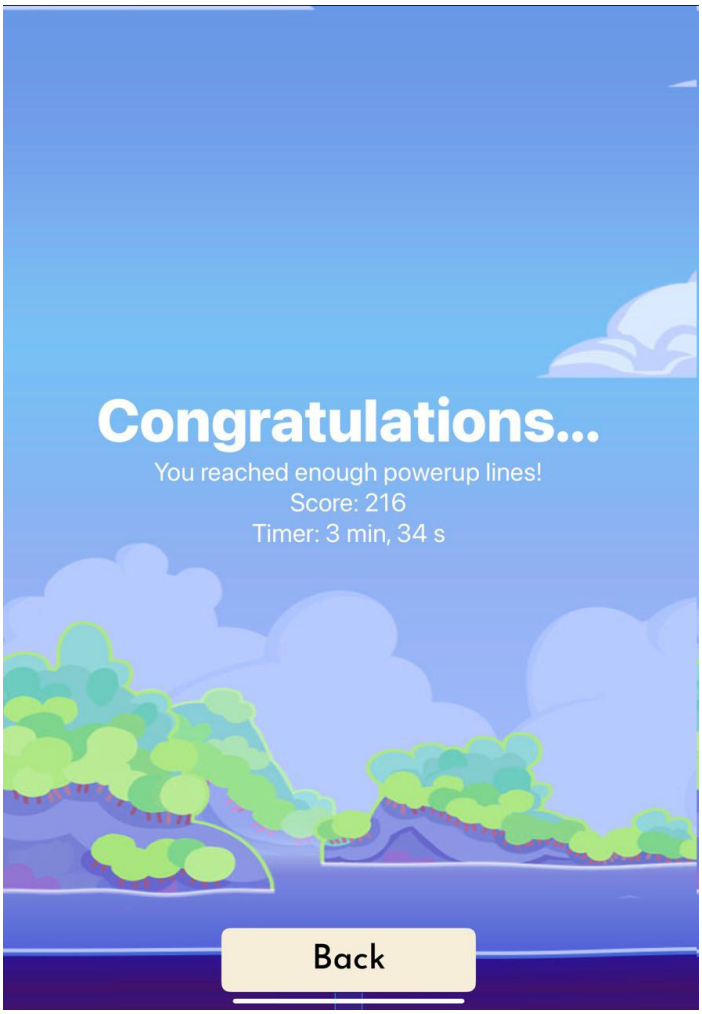


Achievements Screen

The user can view what achievements they have obtained so far while in Singleplayer. The achievement progress can be reset in the Settings Screen.

Game Win Screen

If the user wins, they will see a screen similar to the following:

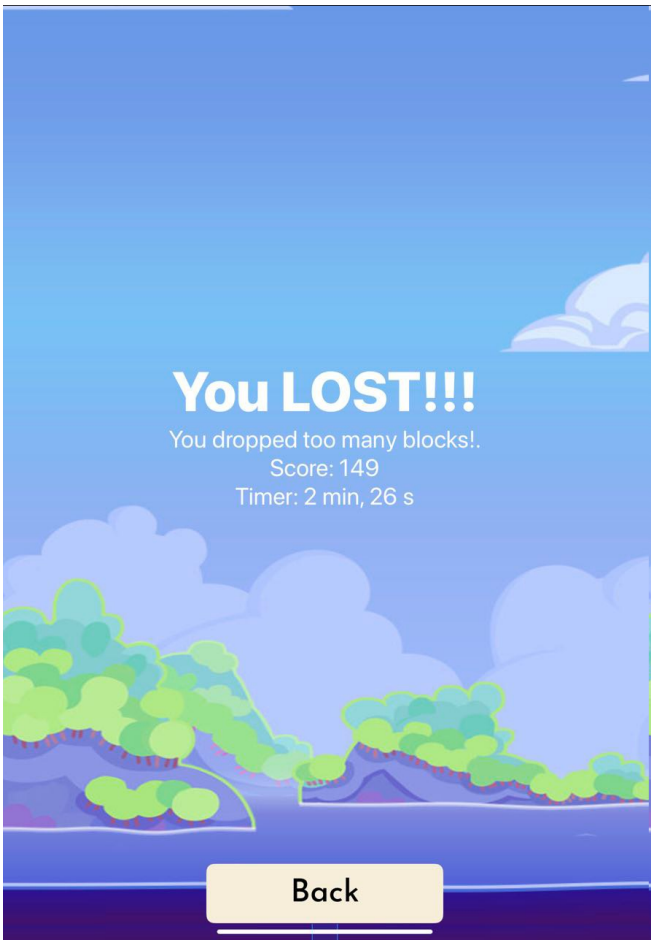


Game Win Screen after winning Scale the Heights game mode

The message displayed is dynamic and changes according to the game mode and game events.

Game Lose Screen

If the user loses, they will see a screen similar to the following:

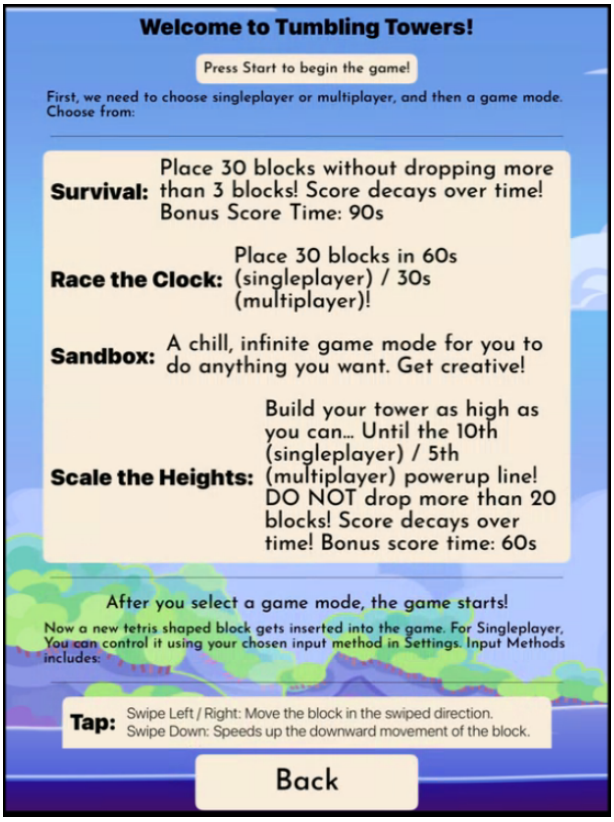


Game Lose Screen after losing Survival game mode

The message displayed is dynamic and changes according to the game mode and game events.

Help Screen

If the user taps the “?” button on the Main screen, they will be directed to the Help Screen which contains a tutorial of how to play Tumbling Towers.

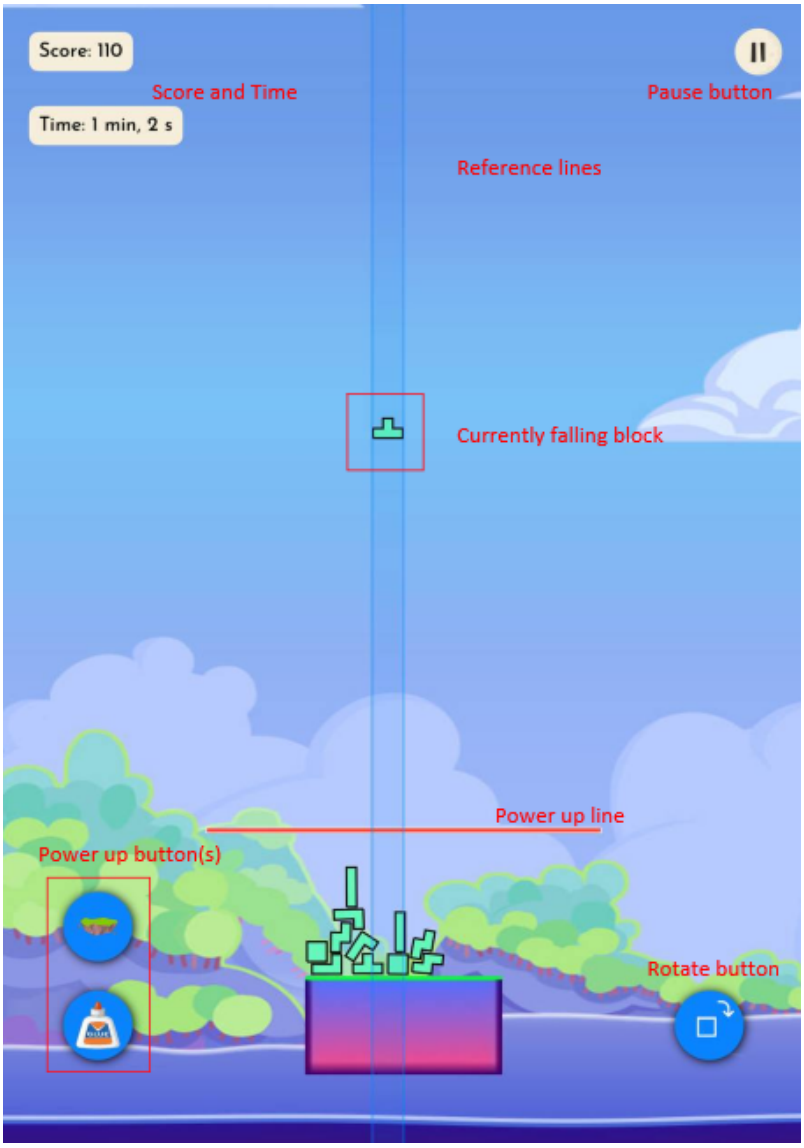


Help Screen

The user can view information about the game such as the instructions for each game mode and the different types of input available on this page.

How to Play

Below is an enlarged view of the Gameplay Screen with the respective labels:



Labeled Gameplay Screen

Horizontal Movement

To move the block left/right, users can tilt the device left/right (if input mode is Gyro) or slide their finger left/right (if input mode is Tap). There are invisible bounds slightly away from the platform that prevent the block from going too far left/right.

Rotate Button

To rotate the currently falling block, tap the button at the bottom right of the Gameplay screen. The block will rotate by 90° clockwise every time the button is tapped.

Downward Movement

To speed up the descent of the block, users can slide their finger downwards.

Power Up Line

The power up line is the way to obtain power ups. To trigger the power up line, build a tower that crosses the power up line. The tower has to be stable (not shaking) for the power up line to be triggered. Upon being triggered, a random power up will be awarded to the user. At any point in time, a user can have up to 5 power ups. If a user currently already has more than 5 power ups, and triggers the power up line, no new power up will be awarded to the user.

Power Ups

To activate a power up, tap any of the power up buttons at the bottom left of the screen. The one which was tapped will get used up.

Score and Time

The current score and the current time elapsed/the time remaining (depending on game mode) will be displayed.

Pause Button

To pause the game, press the pause button.

Game Modes

There are 4 available game modes: Survival, Sandbox, Race the Clock and Scale the Heights.

Survival

The aim of the Survival game mode is to place down 30 blocks on the screen without dropping more than 3 blocks. Only the blocks that did not fall off the platform after they were placed will be counted.

The score of the user decays over time, i.e. the faster the user wins the game, the higher their score will be. If the user wins within 90 seconds, they obtain a bonus to their score.

Sandbox

Sandbox mode allows the user to play the game without any win or lose condition. The user can use this mode to practice or make creative structures.

Race the Clock

In Singleplayer, the user has to place 30 blocks in 60 seconds to win, otherwise the user loses. In Multiplayer, the user has to place 30 blocks in 30 seconds to win.

Scale the Heights

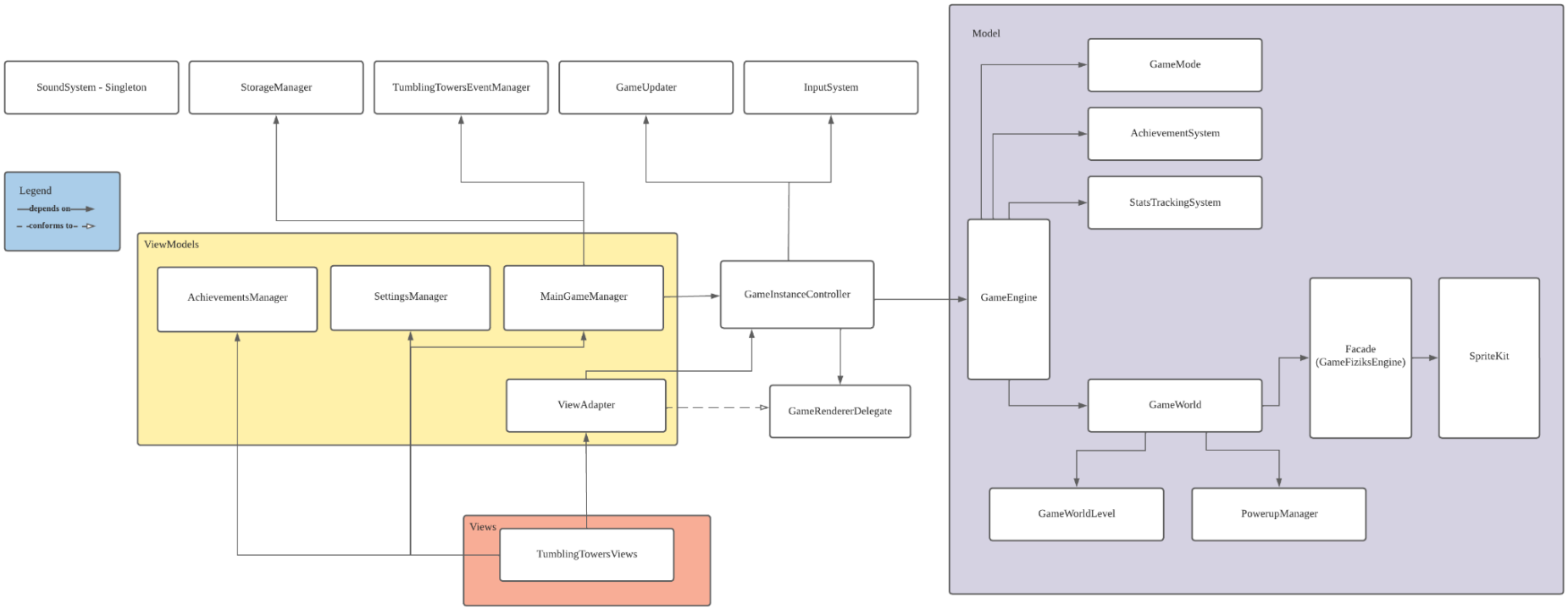
In Singleplayer, the user has to build the tower up to the 10th power up line to win. In Multiplayer, the user has to build the tower up to the 5th power up line to win. If the user drops more than 20 blocks, the user loses.

The score decays over time, and if the user wins within 60 seconds, they get a score bonus.

Designs

Top Level Organisation

The diagram below represents the top level organization of the application.



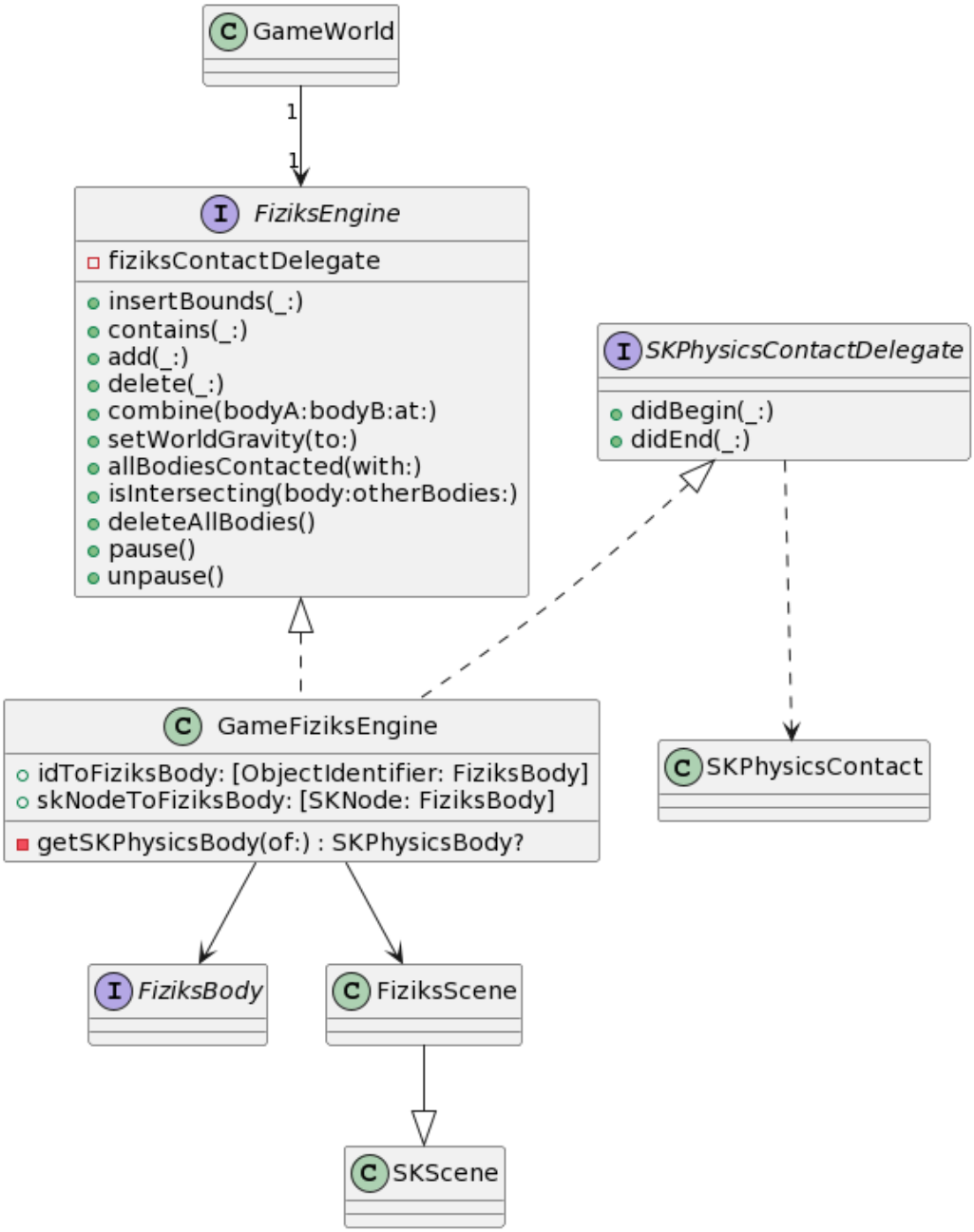
Fiziks

Fiziks is designed to be a standalone local package capable of being used by any application that requires a physics simulation. The Facade pattern is employed so as to allow flexibility in the choice of the subsystem physics engine, not requiring the client code to make any changes should there be any changes to the subsystem of choice.

Module Structure

The module structure for the physics engine will be broken down into two parts, FiziksEngine and FiziksBody.

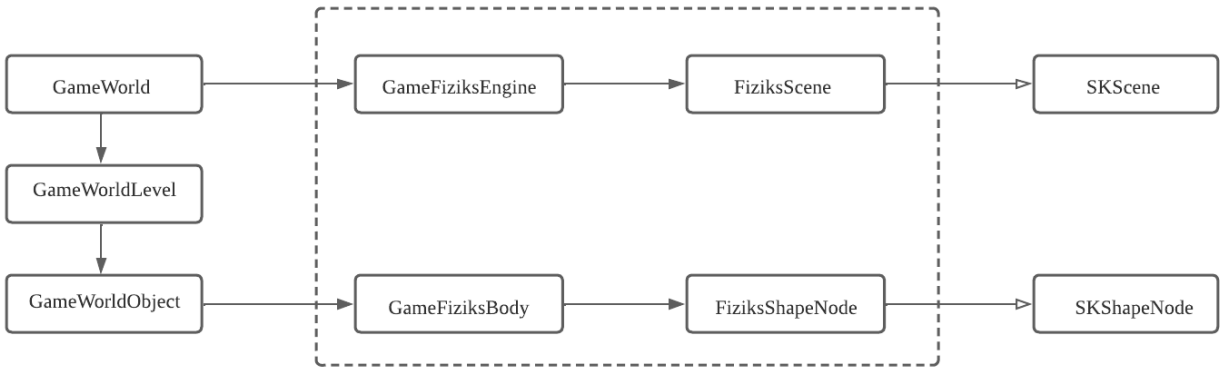
FiziksEngine



Should there be a need to replace the subsystem physics engine, the new physics engine just needs to conform to the FiziksEngine protocol. This also allows swapping of the physics engine, e.g. having a GameFiziksEngine for production, TestFiziksEngine for testing purposes.

In the application, the physics simulation is carried out by SpriteKit (SK). SK comes with many features for game development, but our goal is to use it purely for its physics simulation capabilities. With that in mind, the Facade pattern was used to decouple SK from the rest of the application.

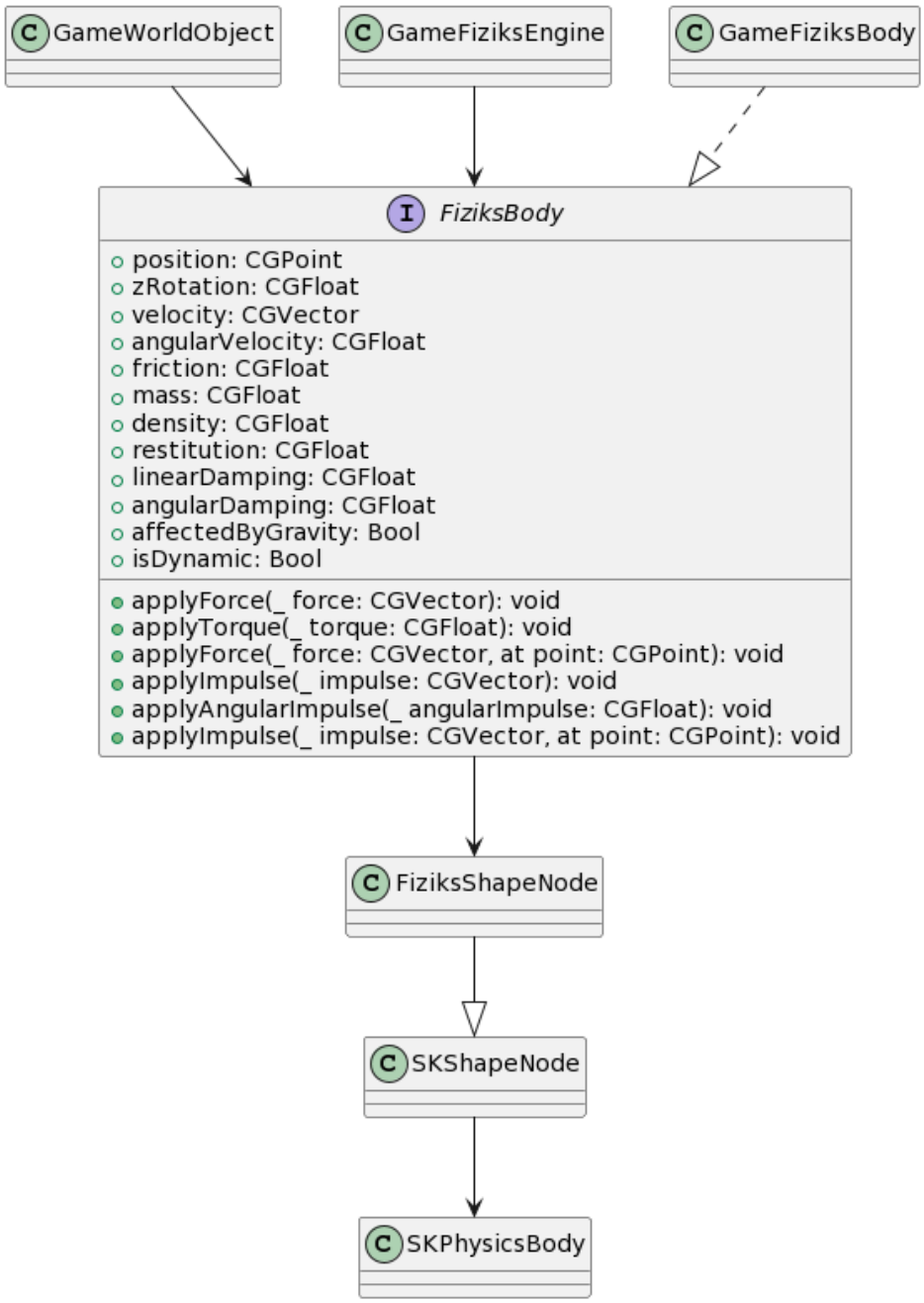
The FiziksEngine represents a protocol defining the methods that must be implemented by the physics engine. We implemented the GameFiziksEngine class, which conforms to the FiziksEngine protocol. This applies the Dependency Inversion Principle, decoupling the implementation of the FiziksEngine from its usage.



As shown in the above simplified diagram, GameFiziksEngine serves as a facade over SKScene while GameFiziksBody serves as a facade over SKShapeNode. **The one exception in this facade is the SKScene which is presented by the GameplayLevelView**. The reason for this is that SK does not allow the physics simulation to run if the SKScene is not presented on some View. The solution is to provide a View for the SKScene to be presented on, but at the same time rendering another view on top of it.

The GameFiziksEngine also conforms to SKPhysicsContactDelegate protocol in order to receive physics contact updates from SK's physics engine. FiziksContact and FiziksContactDelegate are created for the FiziksEngine to send out contact updates in a similar fashion.

FiziksBody



As described above, FiziksBody serves as a facade over SKShapeNode. In fact, it is also a facade over SKPhysicsBody. The reason is that SK stores properties such as position and zRotation in the SKShapeNode while storing other physical properties such as velocity and restitution in the SKPhysicsBody. Since we want to use SK purely for its physics simulation, FiziksShapeNode becomes the one class that holds all information required to simulate a physics environment.

FiziksBody stores a reference to FiziksShapeNode, which subclasses SKShapeNode. Under the hood, SK uses SKShapeNode for the physics simulations. FiziksShapeNode can be seen as the “one source of truth”. We can get and set its attributes as shown below:

```
var position: CGPoint {
    get {
        fiziksShapeNode.position
    }
    set {
        if newValue != position {
            fiziksShapeNode.didUpdatePosition(to: newValue)
        }
    }
}

var zRotation: CGFloat {
    get {
        fiziksShapeNode.zRotation
    }
    set {
        if newValue != zRotation {
            fiziksShapeNode.didUpdateZRotation(to: newValue)
        }
    }
}
```

Runtime Structure

Receiving Physics Updates

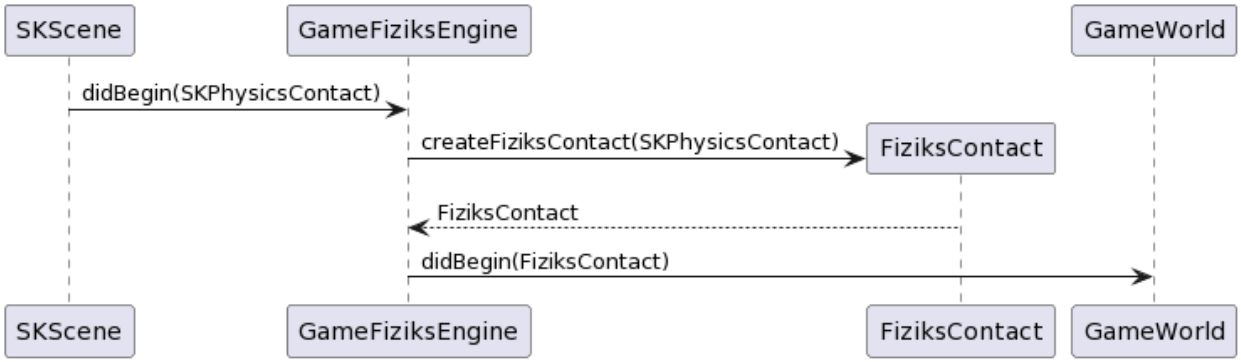
Since GameFiziksBody simply gets its attributes from its FiziksShapeNode, whenever an attribute's getter is called, the most updated value will be taken from the FiziksShapeNode.

Making Changes to FiziksBodies

When the client sets an attribute in the GameFiziksBody, the setter automatically calls an update function in FiziksShapeNode which will perform the necessary updates, allowing SK to continue the physics simulation with the new value.

Contact Begins/Ends

The diagram below illustrates how a contact in GameFiziksEngine is handled, more specifically when a contact begins:



GameFiziksEngine conforms to SKPhysicsContactDelegate. Using the SKPhysicsContact, GameFiziksEngine creates a FiziksContact and calls didBegin() with the newly created FiziksContact object on GameWorld, which conforms to FiziksContactDelegate. For the follow up in what occurs in Game World, refer to [Placing a block](#).

Note that the existence of FiziksContact allows us to decouple the GameWorld from SpriteKit, as the GameWorld does not need to deal with SKPhysicsContact (and SKNode, which is contained within SKPhysicsContact).

When the contact ends, didEnd() is called instead and a similar process occurs.

Design Considerations

FiziksBody

One key consideration is how to propagate updates from the subsystem physics body to FiziksBody and vice versa.

Option 1: FiziksEngine holds a mapping between FiziksBody and FiziksShapeNode
At every tick of the subsystem physics engine, FiziksEngine will loop through all FiziksShapeNodes and copy over all attributes to the respective FiziksBody. Similarly, whenever the client updates the FiziksBody, FiziksEngine is responsible for updating the respective FiziksShapeNode.

This implementation requires a large amount of redundant computation to copy values over and over, even when they are unchanged. As the number of FiziksBodies in the simulation increase, this is likely to lead to performance issues.

There is also the added disadvantage that the code for the updates have to be manually added and is error prone as each attribute has to be copied over one by one.

Option 2: FiziksShapeNode is the "one source of truth" and FiziksBody gets and sets from there
<u>This is the chosen implementation.</u> It is described above.

This implementation has the benefit of reducing the computation of copying over values at every tick of the physics engine.

Additionally, if there is a need to add an attribute to FiziksShapeNode, only FiziksBody needs to have a new computed variable with a getter getting its data from the FiziksShapeNode and its setter setting the new value to the FiziksShapeNode. If **Option 1** was selected instead, there would be an additional step of adding the code to copy the value back and forth inside FiziksEngine.

Thus this option enforces SRP better than **Option 1**.

Option 3: FiziksShapeNode and FiziksBody delegate the updates to each other
Whenever updated, both FiziksShapeNode and FiziksBody would update the other through the delegate pattern.

This is similar to **Option 2**, except that there is a back and forth flow of data.

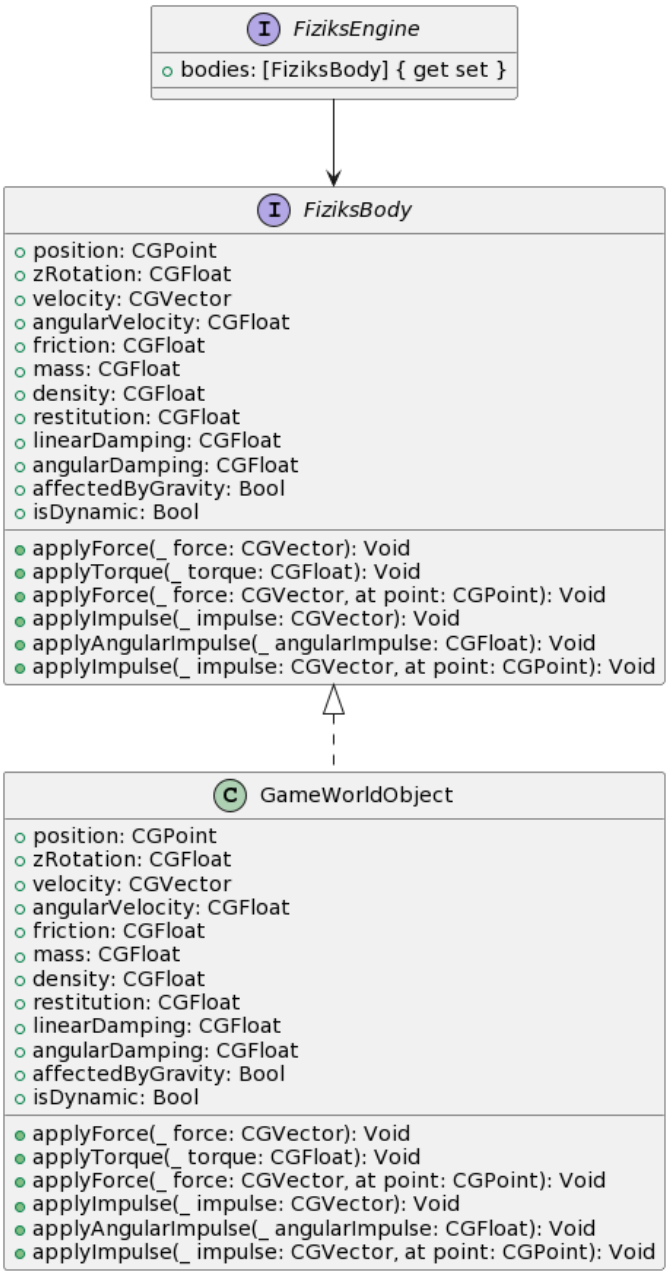
The advantages and drawbacks of this option are also similar to **Option 2** except that due to SK implementation, SKShapeNode does not call its didSet property observer when its attributes are updated. This means that back-and-forth delegates are actually not possible to implement.

One way to circumvent this problem is to use a manually implemented observer pattern to check for a change in the SKShapeNode attribute at every tick and update the FiziksBody if there is a change. However, this solution's drawbacks would be the same as those in **Option 1**.

Interaction between the Model Object and FiziksBody

Another key consideration is how we would synchronize these changes between the FiziksBody and the model object (GameWorldObject).

Option 1: GameWorldObject conforms to FiziksBody protocol

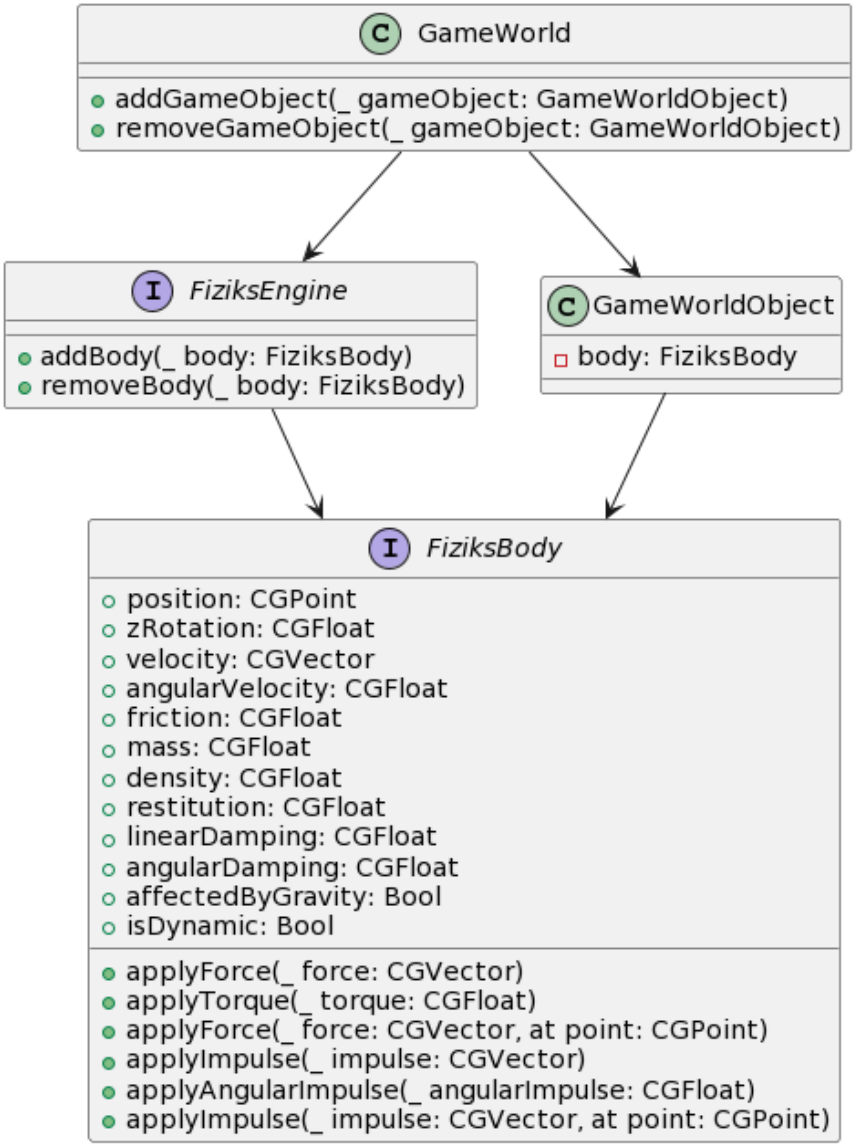


This would allow GameWorldObject to be inserted into FiziksEngine and be directly updated by FiziksEngine (position, other attributes).

The advantage is that attributes like position and zRotation do not need to be copied over from some mapping between GameWorldObject and FiziksBody.

However, this compromises the SoC principle, where there would be no clear separation between the physics engine and the model. GameWorldObject which is part of the model, would have unnecessary physics attributes & methods within itself. Ideally we would want a structure where the game world and the physics engine are not the same component.

Option 2: GameWorldObject stores a reference to FiziksBody



In this implementation, the GameWorldObject stores a reference to FiziksBody. The advantage of this is that in order to obtain attributes of the object such as position and zRotation, we can use computed properties:

```
var position: CGPoint {
    fiziksBody.position
}

var zRotation: CGFloat {
    fiziksBody.zRotation
}
```

Computed Properties of position and zRotation attributes

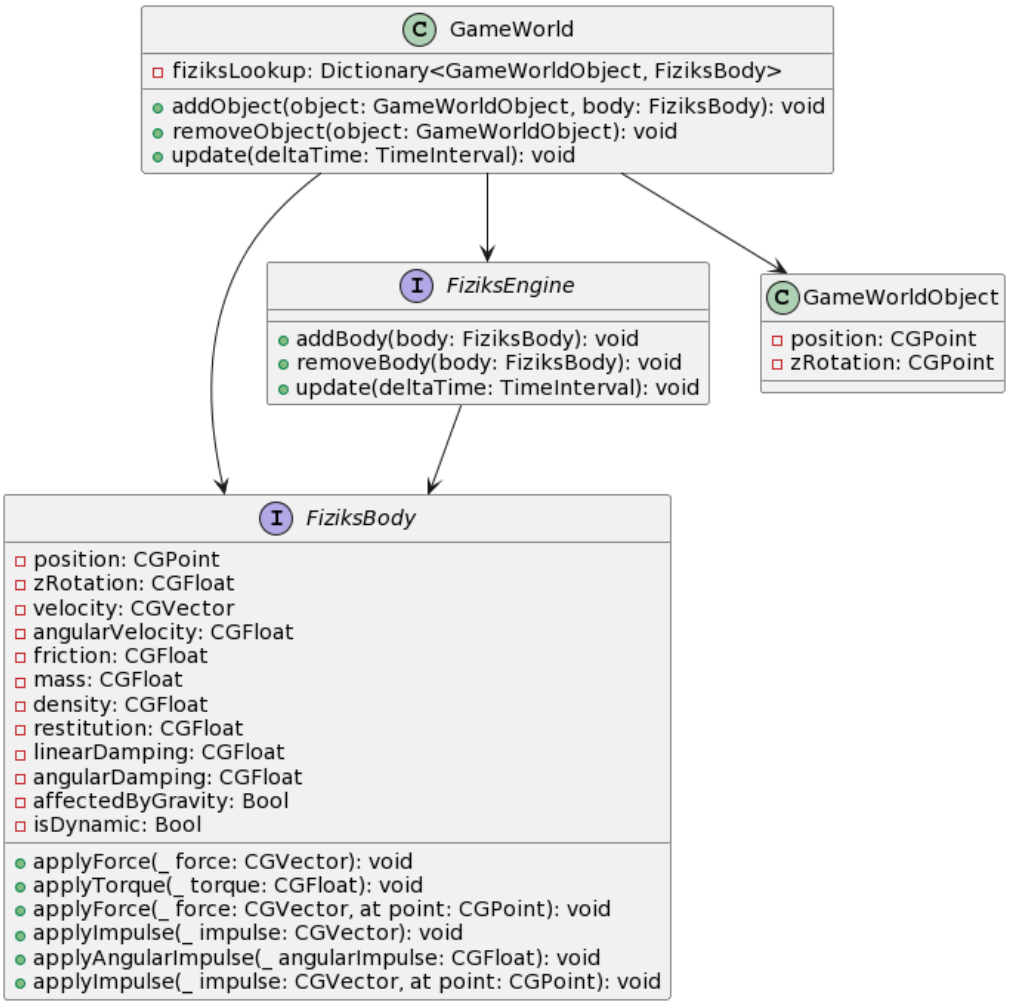
Thus, we do not need to copy the attribute values from some mapping between GameWorldObject and FiziksBody to synchronize the changes, and within GameWorldObject, we have the ability to choose which physics properties to be exposed.

This would add some coupling between GameWorldObject and the FiziksBody. However, this coupling would be justified by the fact that:

1. FiziksBody represents the physics body of the GameWorldObject, so it would not be unusual to store a reference from the GameWorldObject.
2. The FiziksBody protocol type is used, so as to ensure that if a different implementation of FiziksBody is used, the system would perform exactly the same way.

<u>This option was ultimately chosen and implemented.</u>

Option 3: GameWorld stores a mapping between GameWorldObject and FiziksBody



In this implementation, GameWorld would need to store a lookup table to map between GameWorldObject and FiziksBody.

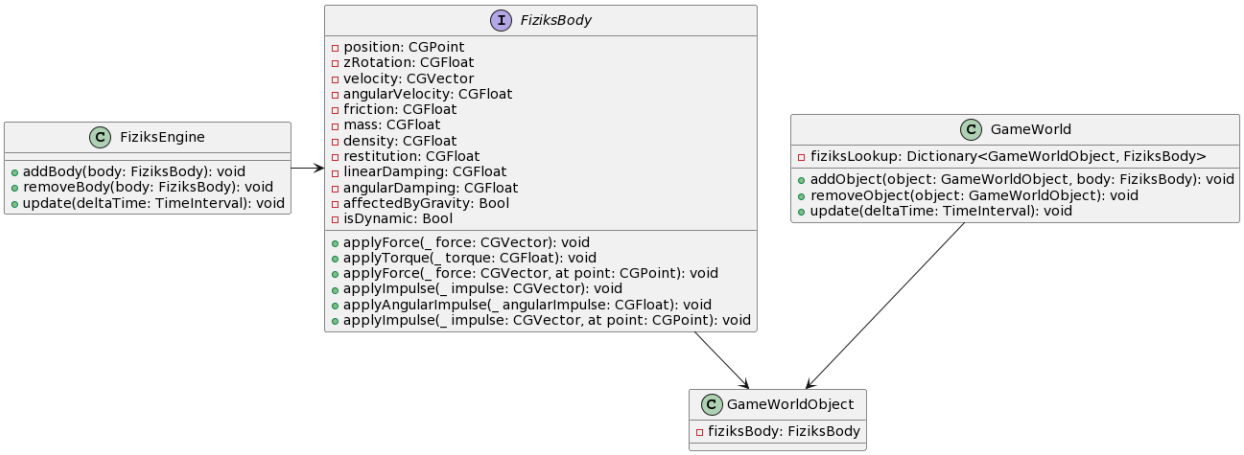
The advantage is that GameWorldObject has no knowledge about FiziksBody whatsoever, decoupling the GameWorldObject and FiziksBody.

The disadvantages are:

1. In order to update an attribute of GameWorldObject, GameWorld needs to find the respective FiziksBody and make changes to it.
2. Every tick of the FiziksEngine, GameWorld would need to loop through all FiziksBodies and update their respective GameWorldObjects.

This compromises SRP, since GameWorld now has additional responsibility of synchronizing the changes between FiziksBody and GameWorldObject.

Option 4: Reference from FiziksBody to GameWorldObject



In this implementation, FiziksBody stores reference to GameWorldObject.

The advantage of this is that FiziksBody would not need to store the attributes position and zRotation, and there is no need to sync these attributes' values between FiziksBody and GameWorldObject.

The disadvantage is that it couples the physics engine with the GameWorld, and since the physics engine should be standalone, this solution **would not be viable**.

Further Discussion

The facade pattern used in the FiziksEngine should allow any subsystem physics engine to be used. In this section, we will discuss the changes required in order to use XXXPhysicsEngine instead of SpriteKit.

FiziksShapeNode

Let us assume XXXPhysicsEngine has a XXXPhysicsBody, which represents a physics body. First, FiziksShapeNode would be changed to inherit from XXXPhysicsBody. Any logic to get and set physics attributes from XXXPhysicsBody should be placed here, e.g.

```
var velocity: CGVector? {
    get {
        physicsBody?.velocity
    }
    set {
        physicsBody?.velocity = newValue ?? FiziksConstants.defaultVelocity
    }
}
```

And physics-related methods should be implemented, e.g.

```
func applyForce(_ force: CGVector) {
    physicsBody?.applyForce(force)
}
```

FiziksBody

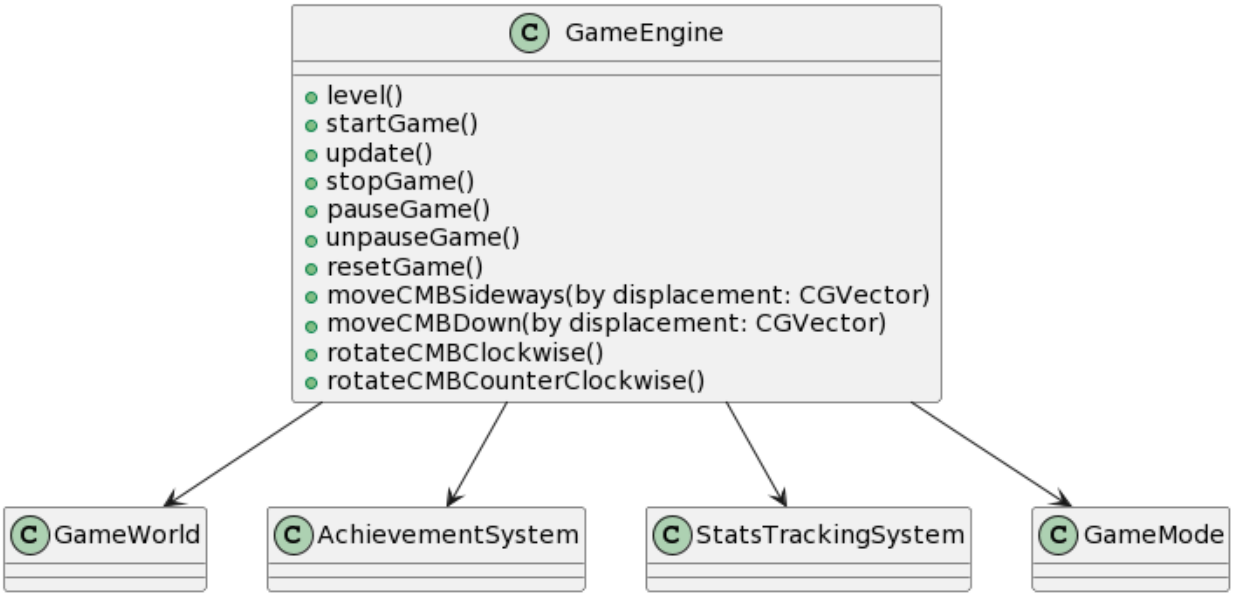
FiziksBody would need to be changed similarly to utilize the attributes and methods exposed by the new FiziksShapeNode.

FiziksEngine

Lastly, GameFiziksEngine would be edited to conform to the new XXXPhysicsEngineDelegate and all methods present in the FiziksEngine protocol would need to be changed in order to maintain the expected functionality.

Game Engine

Module Structure

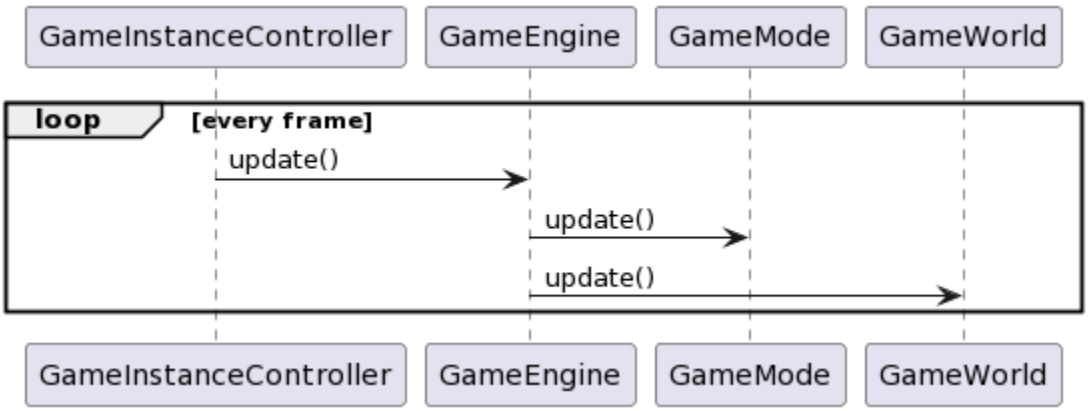


The GameEngine serves as the entry point to the model.

It has functions which allow control over the GameWorld, functions such as startGame(), rotateCMBClockwise().

In addition, it is responsible for instantiating the AchievementSystem and StatsTrackingSystem.

The update() function is called every frame by GameInstanceController, and it calls update() on GameMode and GameWorld to get them to update as well.



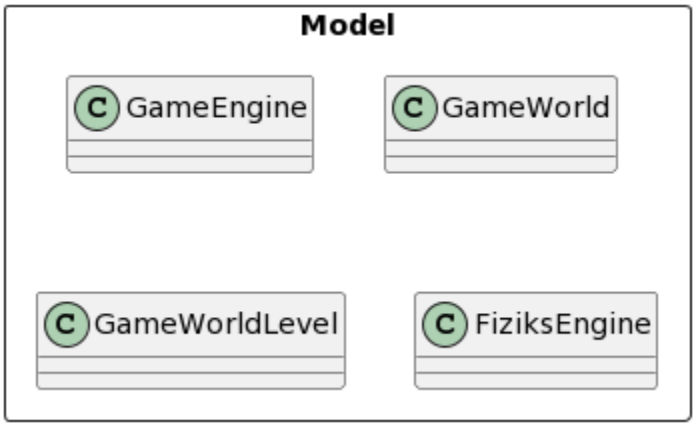
Design Considerations

God Class

For the model, there are two options on how we could have done it.

Option 1: GameEngine + GameWorld + GameWorldLevel
In our initial implementation, our current GameEngine, and GameWorld were in a single class. The original class exposed the level objects to the GameInstanceController to be rendered. However, as we added the power up line, the level boundaries, the additional platforms and the reference lines, we realized that the class was turning into a God object.

Conceptually, it would look like this:



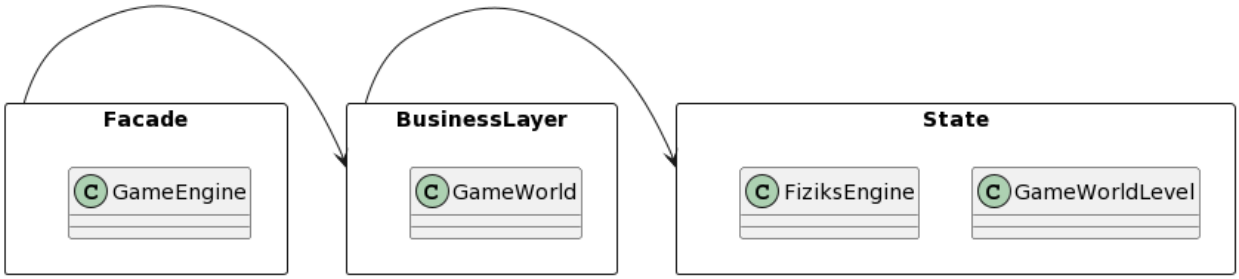
The pros of this approach is that it would have a lower architectural complexity, as there are no abstraction layers.

However, it severely reduces the maintainability of the code, due to the overwhelming number of responsibilities of the original class, which caused it to not conform to SRP.

Option 2: Separation of layers
Alternatively, we could split the responsibilities as such:

1. GameEngine would serve as the entry point of the model. It would expose methods which would be “allowed by the user”, e.g. rotating the CMB clockwise, moving CMB sideways, moving CMB downwards.
2. GameWorld would carry out updates that need to be done on the level, e.g. removing out of bounds blocks, updating power up line height, etc.
3. GameWorldLevel would store the state of the GameWorldObjects.
4. GameWorld also stores the FiziksEngine, which maintains the state of FiziksBody objects.

As such, we can think of the model as split into layers:



This way, GameEngine serves as the Facade to the entire Model and simplifies the interaction with the Model subsystem.

Choice

We chose **Option 2**. This allowed better SRP and SoC, as each layer has a single responsibility. Given a simpler application, Option 1 might have been possible, but due to the increasing number of objects that we had to manage, it would be better to have a dedicated component store and manage the state of all these objects.

Instantiation of AchievementSystem & StatsTrackingSystem

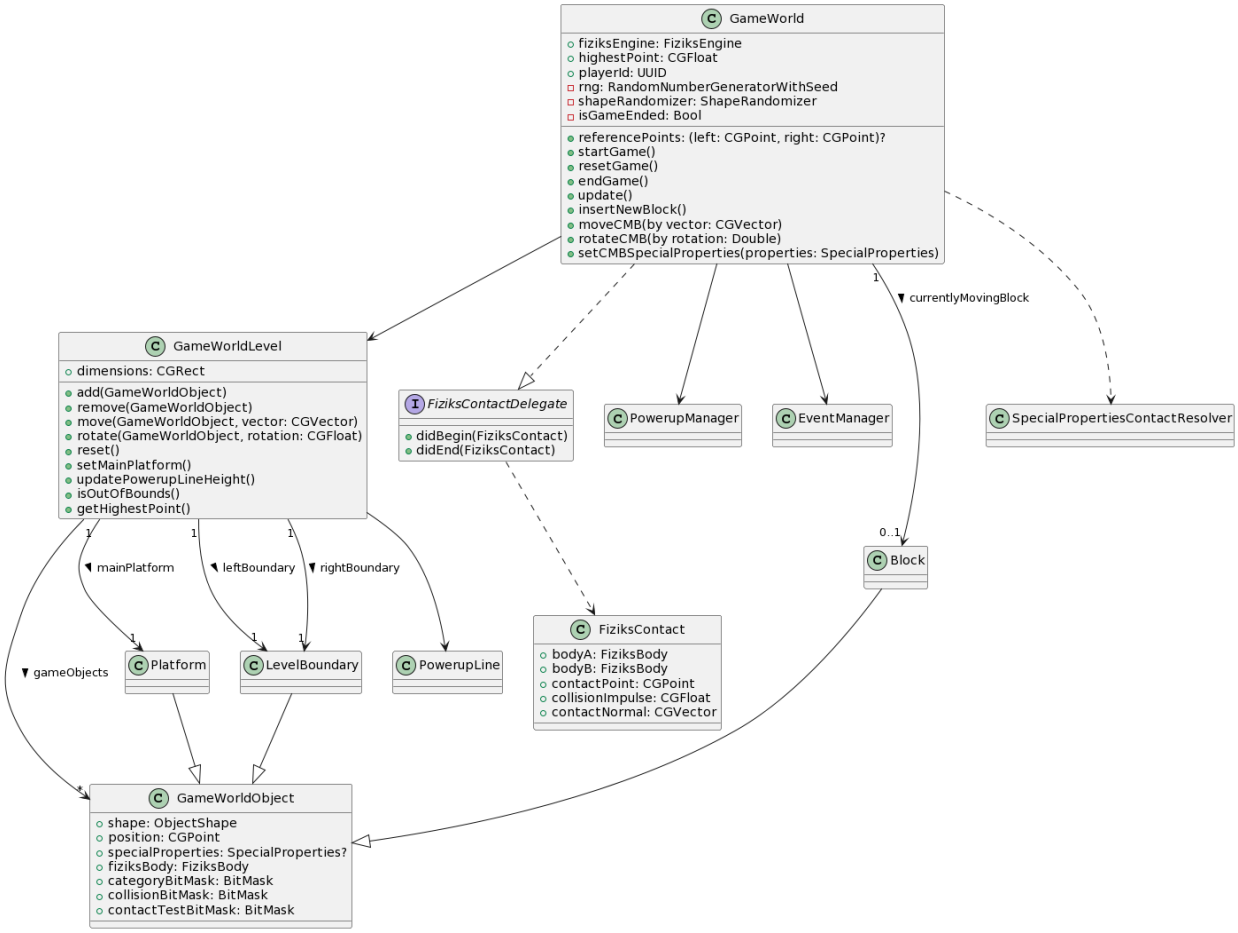
One consideration is whether or not AchievementSystem and StatsTrackingSystem should be instantiated in GameEngine. Since they do not require updates from the GameEngine, nor do they perform any operations on any of the model elements, there is nothing preventing us from instantiating them even outside of GameEngine.

However, the AchievementSystem and StatsTrackingSystem are conceptually a part of the GameEngine as they track data from a specific instance of the GameEngine. When multiplayer mode is implemented, several GameEngines need to be instantiated, and it is reasonable for one to expect that new AchievementSystems and StatsTrackingSystems are instantiated to track the data in their respective GameEngines separately.

Hence, we opted to instantiate AchievementSystem and StatsTrackingSystem inside GameEngine.

Game World

Module Structure



GameWorld serves as the model being managed by the GameEngine. It contains a GameWorldLevel, which stores the GameWorldObjects of the game.

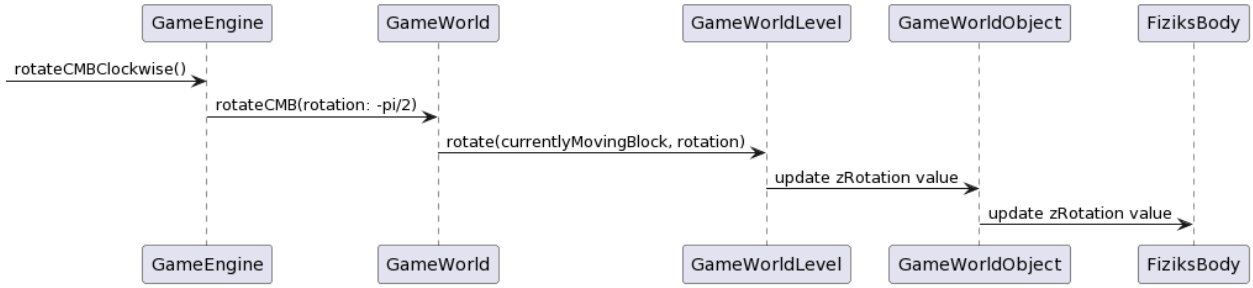
GameWorldLevel stores a reference to a list of gameObjects, which can either be Block, Platform or LevelBoundary.
Blocks are moving, rotatable objects, which are subjected to gravity and physics. Platforms are non-moving and are not subjected to gravity but subjected to physics (in terms of collisions). LevelBoundary are static, invisible objects which prevent the currently falling block from going too far away from the platform.

The currentlyMovingBlock (CMB) is the block that is falling at any point in time, if any. The GameWorld stores reference to the currentlyMovingBlock, in order to be able to control it (by rotating, moving it, etc).

Runtime Structure

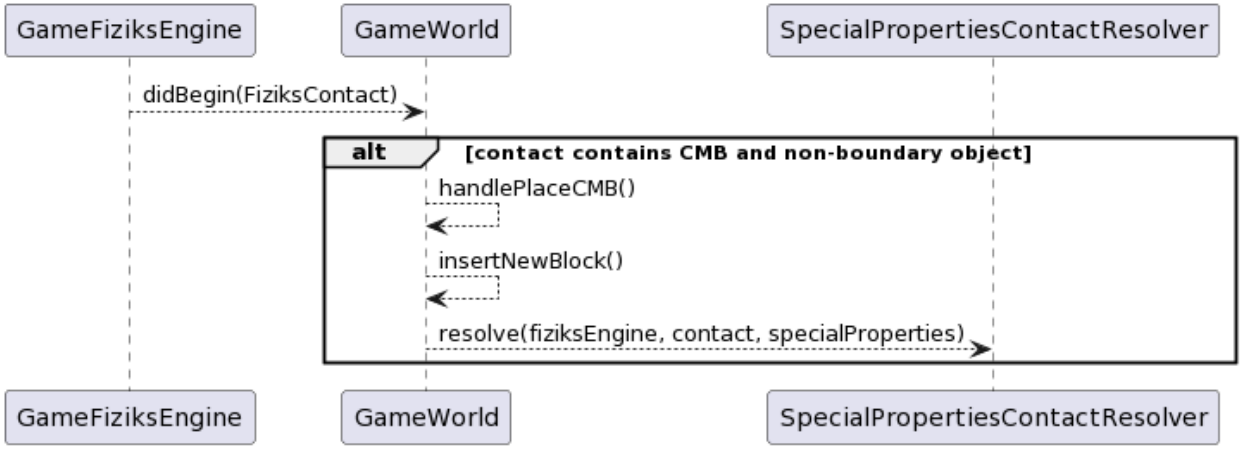
Rotating/Moving a block

The sequence for rotating a block clockwise is as follows:



The sequence for moving a block is largely similar to the above sequence.

Placing a block



When two objects collide, the GameFiziksEngine calls didBegin() on its FiziksContactDelegate, which is GameWorld.

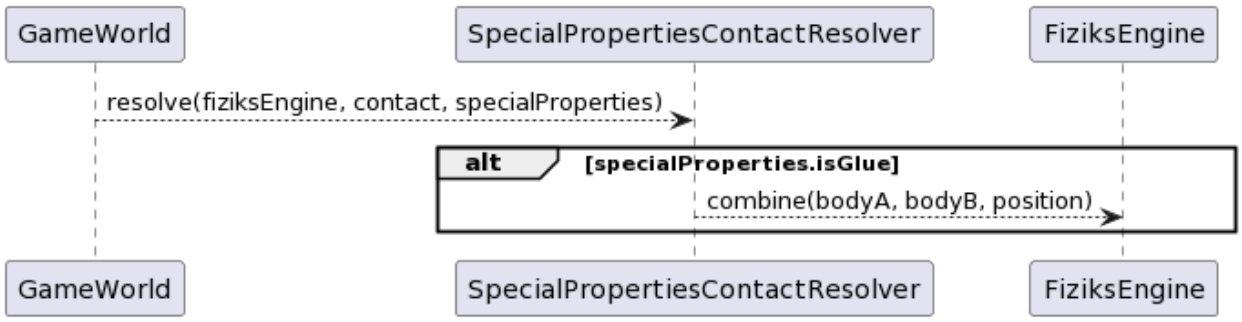
The check ensures that the CMB is part of the contact, and a non-boundary object is part of the contact (we don't want the CMB to be considered placed when it touches the boundary).

When the CMB is placed, a new block spawns at the top of the screen.

The resolve() method is to handle any special effects that should occur when the block is placed (e.g. glue effect).

SpecialPropertiesContactResolver

Sometimes, the CMB might have a special property, e.g. glue, if a power up is activated. SpecialPropertiesContactResolver handles these cases.



Design Considerations

SpecialPropertiesContactResolver

Sometimes, the objects might have special properties. Currently, the only special property is glue (from glue powerup). However, it is easy to imagine that in the future, we might need to add more special properties (e.g. explosive block). As such, a consideration is how to handle these special effects.

Option 1: Store properties in GameWorldObject
One way we can do this is by storing the properties in GameWorldObject as such:

```
protocol GameWorldObject: AnyObject {
    // other properties/methods

    var isGlue
    // other special properties...
}
```

The special effects can then be carried out in the handlePlaceCMB() or didBegin() method.

However, the issue is that this violates OCP, as each time we want to add a special property, we need to modify two unrelated files: GameWorldObject (to add the new property) and GameWorld (to add handling of the new effect).

Option 2: Store properties in SpecialProperties
An alternative way is to store these special properties in a separate entity, SpecialProperties. GameWorldObject would have a reference to it:

```
protocol GameWorldObject: AnyObject {
    // other properties
    var specialProperties: SpecialProperties { get set }
}
```

The handling of these special properties should be abstracted out into another component, SpecialPropertiesContactResolver's resolve() method:

```
class SpecialPropertiesContactResolver {
    static func resolve(fiziksEngine: FiziksEngine, contact: FiziksContact, specialProperties: SpecialProperties) {
        if specialProperties.isGlue {
            fiziksEngine.combine(bodyA: contact.bodyA, bodyB: contact.bodyB, at: contact.contactPoint)
        }
    }
}
```

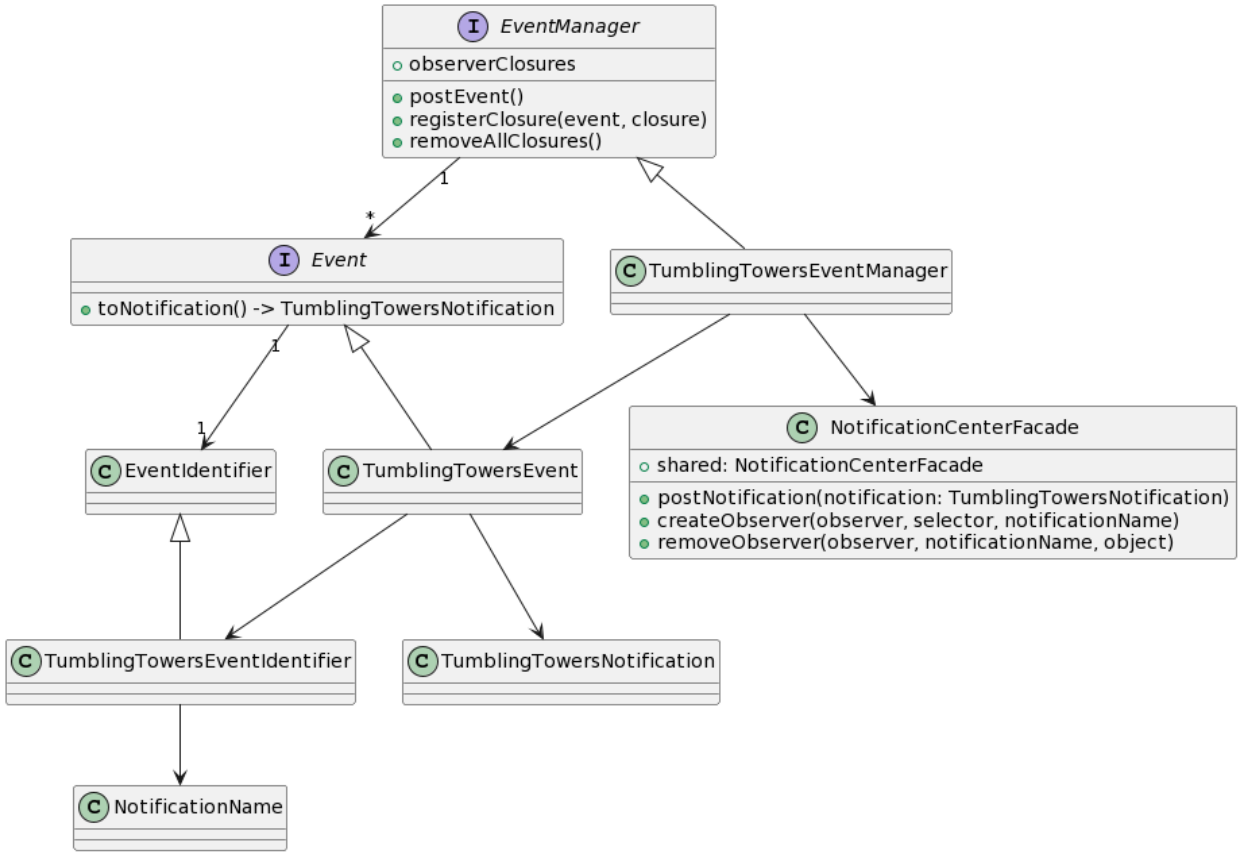
The if-else allows for complex combinations of special properties, e.g. SpecialPropertyA and SpecialPropertyB have some effect, SpecialPropertyA only has some other effect, etc. For now, it is only the glue effect.

This better conforms to OCP, as no unrelated files need to be changed when a new special property is added. The files involved would be SpecialProperties and SpecialPropertiesContactResolver (if we need to add an effect on contact), which is reasonable.

Event Manager

Events were initially considered when we had to work on packages such as GameMode, PowerUp, Achievement, Sound and potentially more in the future development. All of these packages involve processing certain actions when a game event has occurred.

Module Structure



There are 2 protocols, Event and EventManager and 1 class, EventIdentifier which the package has to conform to. The EventManager handles the Event instances by subscribing closures to it. Whenever the Event instance is fired, all the closures are run. Each Event instance also has the associated EventIdentifier

The implementation classes are as follows: TumblingTowersEventIdentier, TumblingTowersEvent and also TumblingTowersEventManager that conforms to the classes / protocol as shown in the class diagram.

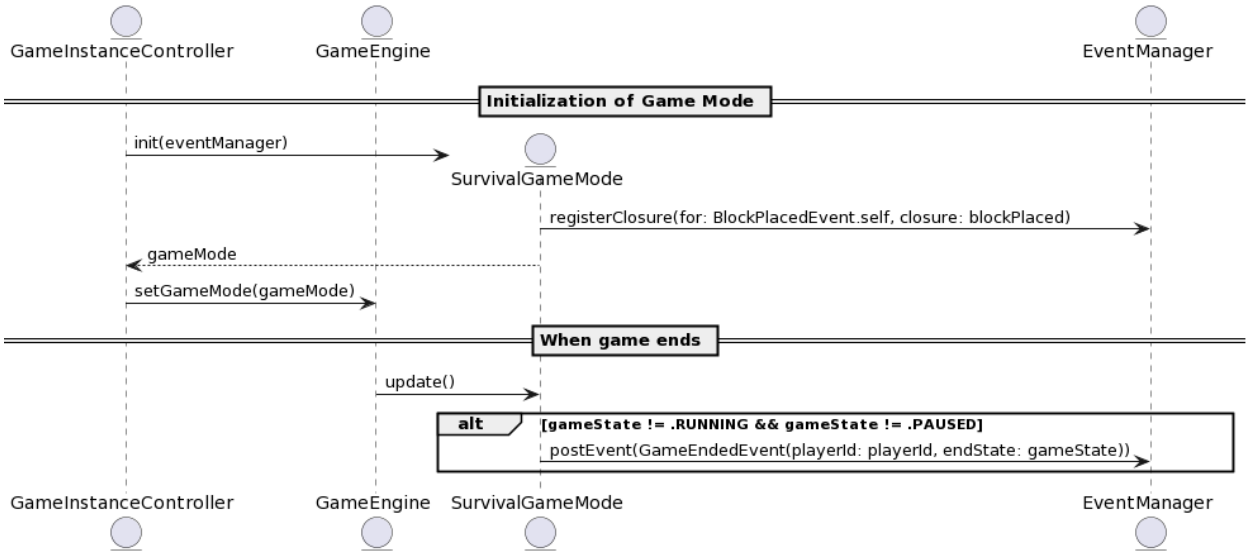
NotificationName, TumblingTowersNotification and NotificationCenterFacade are all wrappers for NSNotificationCenter. In the event that NSNotificationCenter is changed with something else, only these 3 files in the package have to be replaced.

NotificationName is a wrapper around the Notification.Name class from NSNotificationCenter, TumblingTowersNotification is a wrapper around the Notification class from NSNotificationCenter and NotificationCenterFacade is a facade of NSNotificationCenter.

Runtime Structure

Registering & posting events

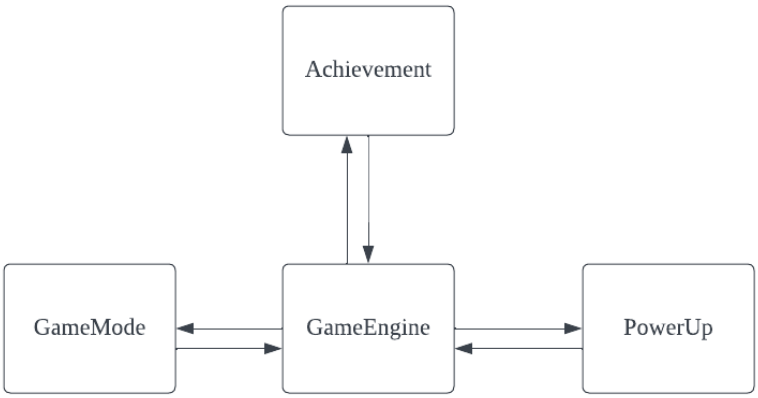
The sequence diagram below shows how the SurvivalGameMode registers a closure for an event during initialization as well as posting an event when the game wins or loses.



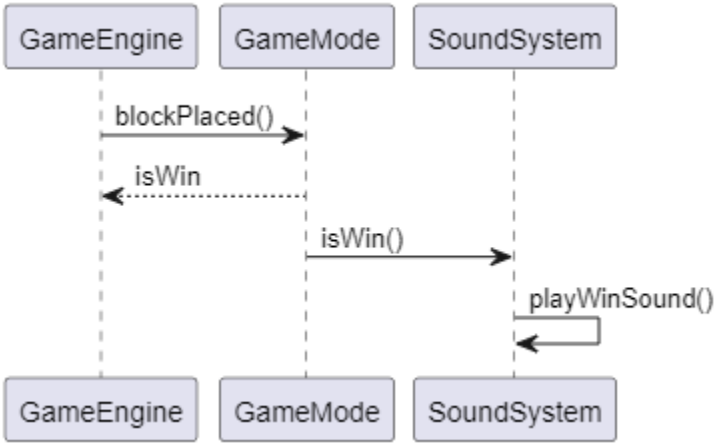
This sequence of registering closure during initialization and posting events is similar in other packages that use EventManager.

Design Considerations

The first consideration was to have a delegate for each package and have GameEngine (the class which processes the game and hence detect game events) to communicate with each package as shown below:

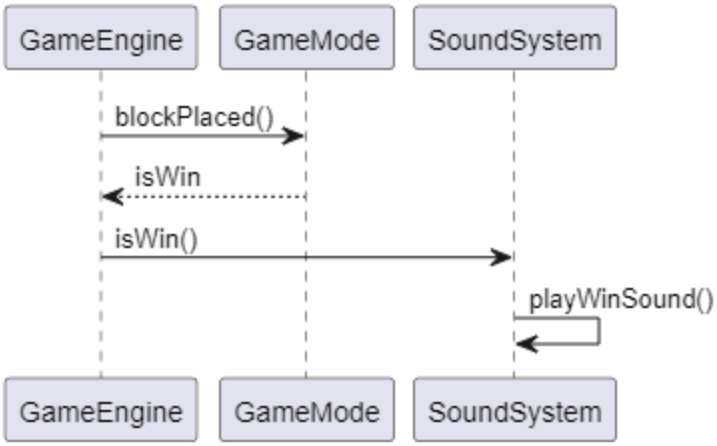


The problem arises when the events in the packages are actually interlinked with each other, as shown in the diagram below:

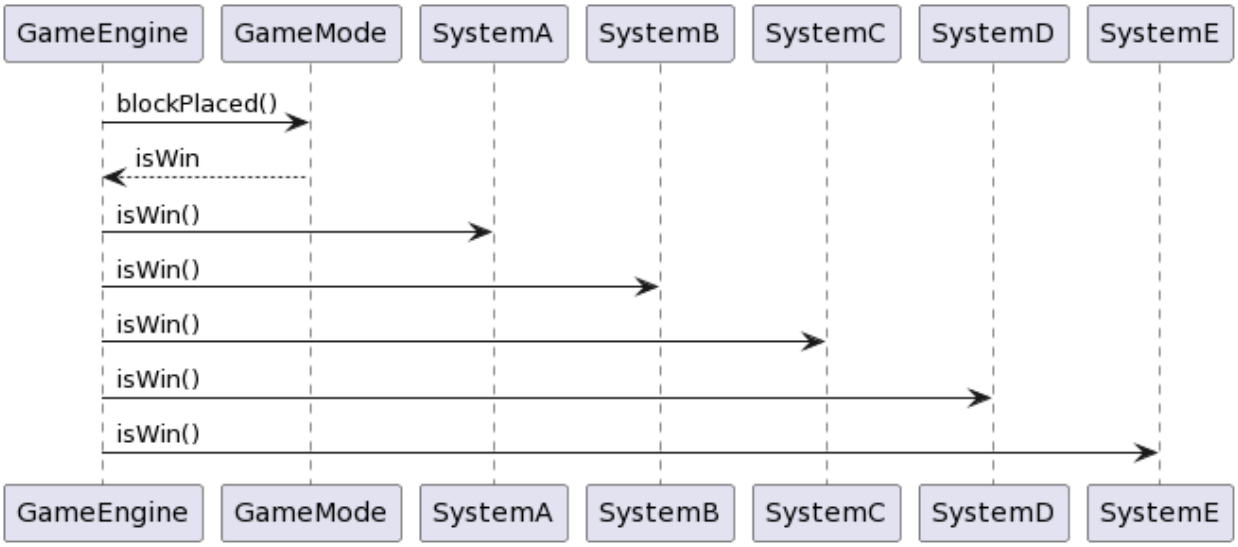


In this example, when a block is placed, the GameMode determines that the game is won and this also affects SoundSystem as it has to play a winning sound to the user.

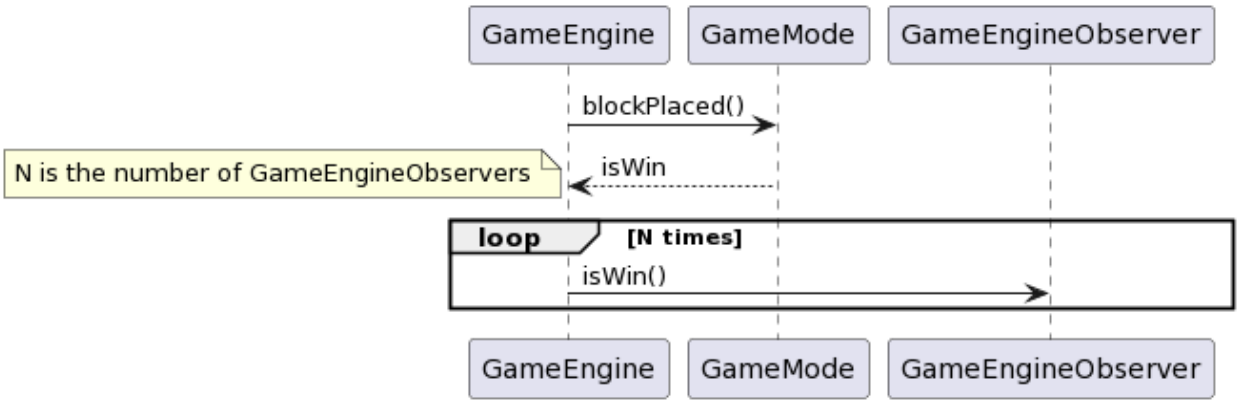
However, this is not ideal as each package should be standalone and not have tight coupling with other packages.
One workaround is to send an event back to GameEngine for it to relay to relevant packages, as shown below:



However, if there is an event that requires updates in various packages, the GameEngine has to call the function in each package, thus resulting in bloating of the code as the number of packages escalates:



This could be mitigated by using Observer pattern, and GameEngine calls isWin() on all its observers:

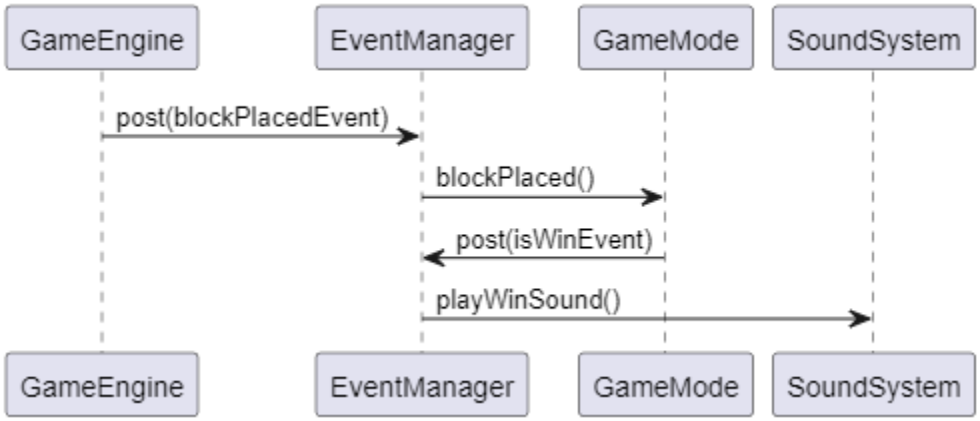


Nonetheless, there are still problems with this.

While there is a looser coupling, the Observers still need to subscribe to the event by registering themselves as an observer of GameEngine. Suppose we call the protocol GameEngineObserver. These packages would need to implement all the methods of GameEngineObserver, even if they do not actually need to observe all the events emitted by GameEngine, resulting in bloating of the codebase. Ideally, we only want the packages to observe events that they are interested in.

In addition, this gives GameEngine an additional responsibility of having to be the one to inform all these packages (albeit it being made simpler via Observer pattern). Ideally, we would want to employ Separation of Concerns and let a dedicated party handle the informing of packages of events that have occurred.

Thus, an EventManager that utilizes the observer pattern (implemented via NSNotificationCenter) is created to act as a mediator between event emitters and observers. When GameEngine and the packages are created, the observers and its closures are registered in the EventManager. When an event is fired, all related closures will be processed. The diagram below shows how the EventManager works in the previous example:



This allows packages to only listen to events that they are interested in, and GameEngine only has to do a single thing - post the event that has occurred.

After deciding on using an EventManager, we considered how to use the EventManager:

Option 1: Event Manager as a singleton
Events would likely be fired everywhere, in models and also in the future, possibly in the view models (e.g. event where the user taps on the screen etc). Due to this, we initially considered implementing EventManager as a singleton.

This adds convenience when developing as well as reduces the need to pass the EventManager from parent component to child component multiple times (similar to prop drilling in React) in the codebase.

However, as the EventManager class has a state that can be modified by other classes, exposing EventManager as a singleton would risk a case where a future developer unknowingly and wrongly modifies EventManager, thus affecting all other classes that use it.

Option 2: Initialize 1 EventManager object and pass it to relevant classes
We can create a single event manager (not a singleton instance) that is passed around. The single EventManager would be instantiated at the main view model to be passed down to the relevant classes. This method increases testability where stubs can be utilized. This option also has the benefit of reducing the exposure of this EventManager instance to other classes that do not use it and hence cannot modify it and cause any unwanted side effects.

However, as previously mentioned, it would require additional code to pass the Event Manager around the different components, increasing the complexity of the codebase.

Option 3: Instantiate 1 EventManager for each package that has an observer

We can create one instance for each package, e.g. PowerUpEventManager, GameModeEventManager etc. This further improves Separation of Concerns of the event managers, in terms of only handling events related to its own package.

However this would cause performance issues due to multiple instances of EventManager.

In addition, the scenario where 1 event that is fired causes changes to 2 packages observers cannot be directly done using this option. The event then has to be fired multiple times to multiple relevant package EventManagers.

A workaround could be a nested structure where when an event is fired, a “main” EventManager relays those events to relevant packages’ EventManager, which adds unnecessary complexity without much benefit.

Choice
We chose **Option 2** due to the additional benefits of increasing testability. It would also prevent the disadvantage mentioned in using a singleton instance, as well as not adding too much complexity or performance degradation unlike Option 3. We also decided that having 1 Event Manager for each package is unnecessary.

Game Modes

The Game Mode is responsible for handling all the logic that could change across game modes. This includes winning and losing conditions, in-game text specific to this game mode, score system, as well as time keeping behavior. It also tracks the conditions of the game that affects the game winning and losing state.

Implementation Goals:

- Simple and fast to create new Game Modes that contain new and different behavior for logic that can possibly change across game modes:
    - Winning and Losing Conditions
    - Score System
    - Timer Behaviour
    - In Game Text (Different Ending Screen Text for different game end states, Game Mode Name and Description)
- Easy to modify behavior of existing Game Mode
- Easy to remove existing Game Mode

Design Considerations

In order to implement the functionality of Game Modes, we considered a few different options:

Option 1: All Game Mode logic handled inside GameEngine and other locations in Model

We can add the code required for the game mode implementation (EG: To check for winning and losing conditions, to modify the score) at the parts of the code that will trigger game mode behavior. This requires many different files to be modified as the logic for different game elements are in different files.

```
private func updatePlacedBlocksStatus() {
    var placedBlockCount = 0
    for gameObject in level.gameObjects {
        if gameObject.fiziksBody.categoryBitMask == CategoryMask.block {
            placedBlockCount += 1
        }
    }

    if gameMode == .SURVIVAL || gameMode == .RACETIME, placedBlockCount > placedBlockThreshold {
        eventManager.postEvent(GameEndedEvent(playerId: playerId, endState: .WIN))

    }

}
```

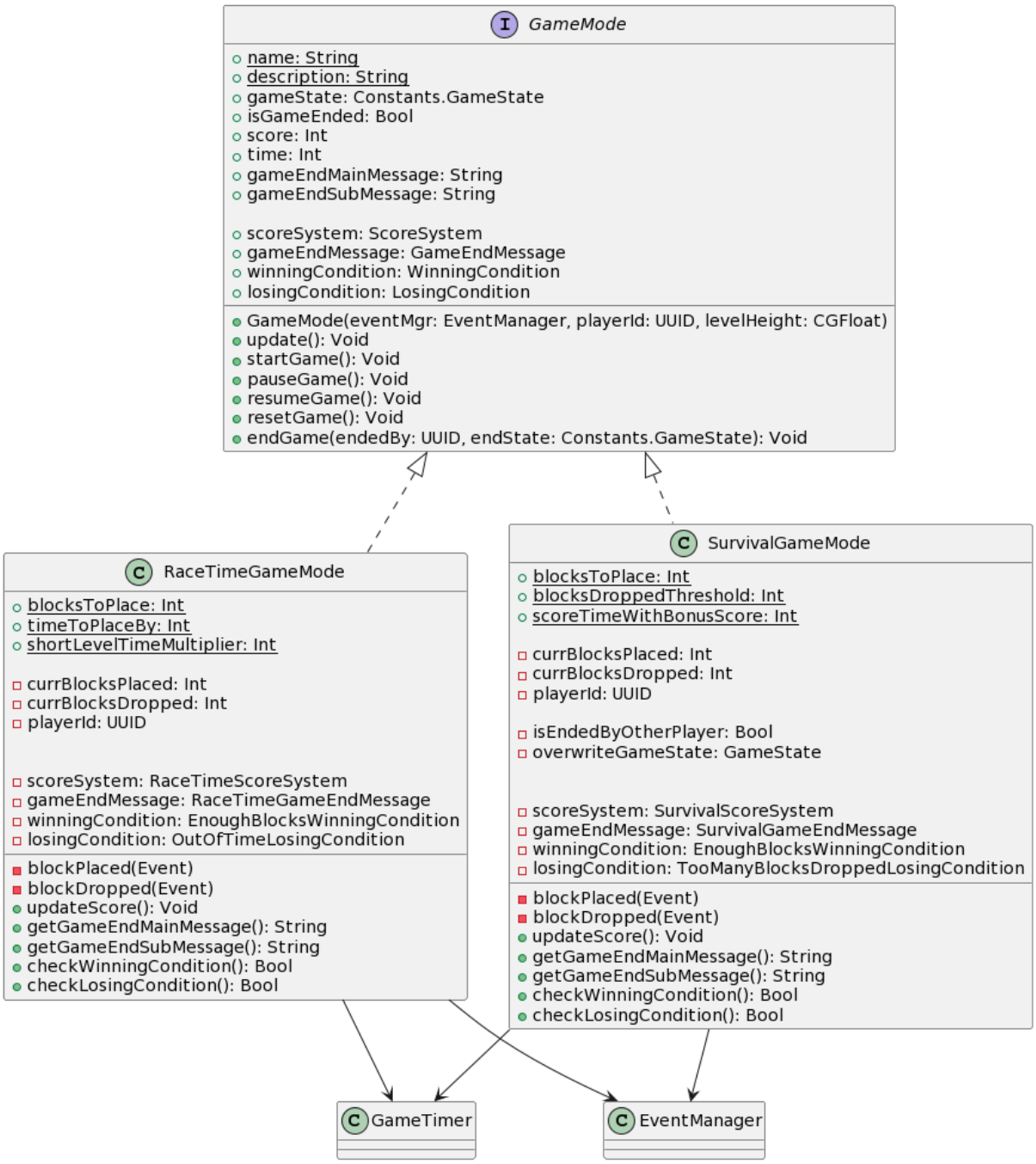
Option 1: Example of a part of Survival Game Mode Logic Implementation inside GameWorld

This approach follows DRY principle as this method could lead to less code when there are some game modes that share common behavior like in the image above, where the same code is reused for 2 different game modes.

However, this is a very naive solution as the logic for a game mode would be scattered throughout the code base. This makes it very difficult for the developer to perform any of the tasks listed under the Implementation Goals, as the developer would have to search throughout the entire code base to create a new game mode, modify or delete an existing game mode, making Game Modes very inextensible under this approach. Many existing classes have to be modified to add a new game mode, violating the Open Closed Principle.

Option 2: Different classes for different aspects of Game Mode (EG: Different ScoreSystem, GameStateTracker etc classes)

We have a GameMode protocol that all game modes implement that contains common functions and properties needed to be accessed by TumblingTowers. This exposes such functions and properties to the rest of TumblingTowers. We can also have additional behavior classes for each different aspect of Game Mode that could change according to the selected game mode.



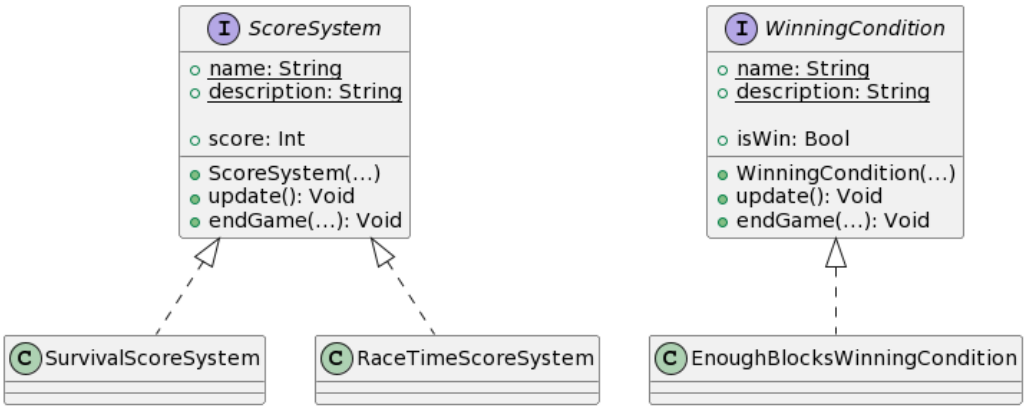
Example of Option 2 with 2 Game Modes

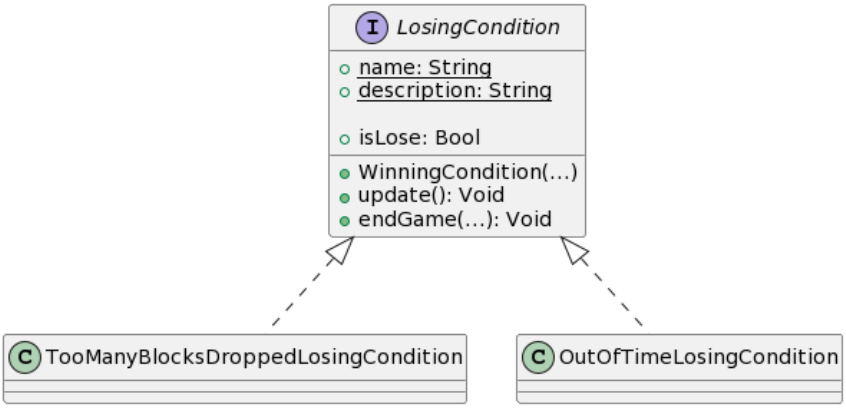
Game Mode Behaviour Classes

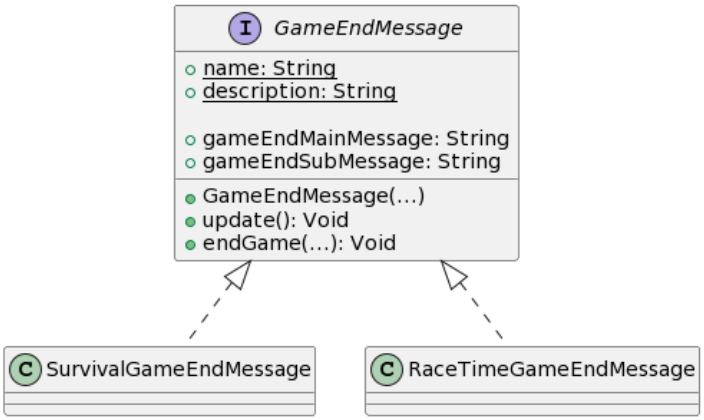
These behavior classes implement common protocols for their respective behaviors.
Such protocols include:

- ScoreSystem Protocol (EG: DefaultScoreSystem, ScaleHeightsScoreSystem)
- GameEndMessage Protocol (EG: SurvivalGameEndMessages, RaceTimeGameEndMessages)
- WinningCondition Protocol (EG: EnoughBlocksWinningCondition, TallEnoughWinningCondition)

- LosingCondition Protocol (EG: TooManyBlocksDroppedLosingCondition, OutOfTimeLosingCondition)







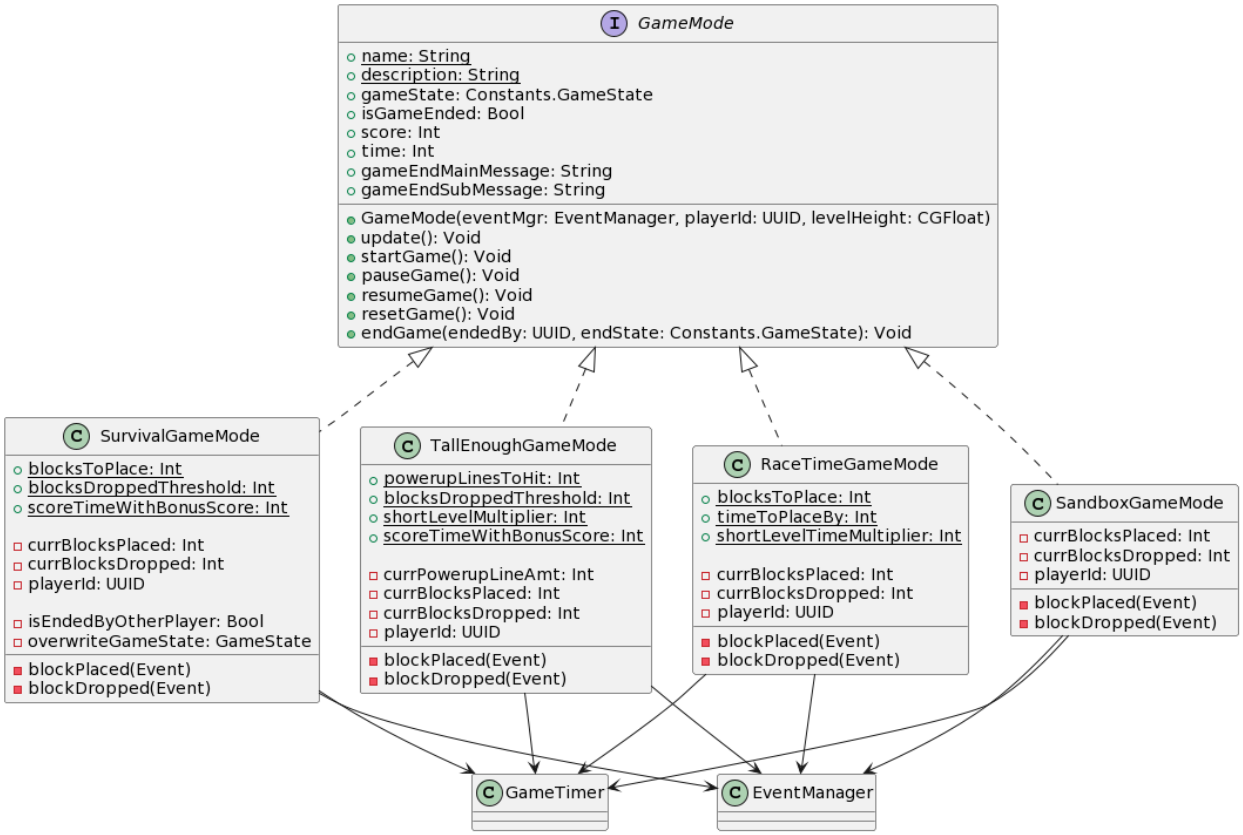
GameMode Option 2 Additional Behavior Classes

This approach follows the DRY principle as common behavior logic is not repeated across gamemodes, which could lead to less code. This approach is good if many of the game modes share common behavior.

However, there are many classes required for this implementation if most game modes have completely different implementations for each behavior.

Option 3: Different GameMode Classes Implementing Game Mode Protocol

We have a GameMode protocol that all game modes implement that contains common functions and properties needed to be accessed by TumblingTowers. This exposes such functions and properties to the rest of TumblingTowers. This is similar to Option 2, except that instead of additional behavior being implemented in their own classes, those behavior that can change across game modes are specified in the game mode file itself.



This approach reduces the number of files that have to be created to only **1** when we want to create new types of game modes. The number of files to modify when changing the behavior of an existing game mode is also reduced to only **1**. This makes it very simple to achieve the Implementation Goals specified earlier. Only the GameMode class has to be touched instead of all the ScoreSystem, GameEndMessage,

WinningCondition, LosingCondition classes that have to be created/modified/removed every time we want to make changes to Game Mode.

However this approach uses less classes, and makes it harder to follow the DRY principle if many game modes contain the same logic, since the same logic would then have to be specified in each game mode class.

Choice

Option 1 is out as it is not extensible and does not follow Open-Closed Principle.

Between **Option 2 and 3**, we chose **Option 3** because the game modes of Tumbling Towers all use behaviors that are different. This is naturally the case as we want to make the different game modes be as different so that the game modes are not repetitive.

As described earlier, **Option 3** also makes it very simple to create new game modes as it minimizes the number of files that have to be created to only 1 for GameMode, compared to **Option 2** where files have to be created for ScoreSystem, WinningCondition etc.

The actual implementation for most of these additional behavior classes are mostly one liner as well, hence the amount of boiler plate code required to create these additional classes (such as init methods and common properties) far outweighs any code savings we could get from some game modes sharing the same logic.

Having less code and less classes makes **Option 3** less confusing and over complicated as compared to **Option 2**. This is a huge advantage as it makes it extremely simple to create a new game mode.

Creating a new Game Mode

In order to create a new game mode, we only need to create 1 file, change 2 lines in the Constants file (to register that gamemode's existence with an enum and dictionary list). No other parts of the code have to be updated even if they use Game Mode due to how we designed Game Mode.

In order to illustrate how simple it is, we look at an example, where we added the Scale the Heights game mode:
1. Add only 1 file that conforms to GameMode protocol. Specify behavior specific to the Scale the Heights game mode within that new file.

2. Add Scale the Heights to GameModeTypes enum and gameModeTypeToClass dictionary in Constants file to register this gamemode's existence to TumblingTowers. This is a 2 line modification.
3. Add a button in GameModeSelectView to allow users to select this gamemode.

After this, other areas of TumblingTowers will know about the new game mode's existence and change accordingly. For example, the tutorial on the Help Screen auto updates to display help information about the new Scale the Heights game mode, even though neither the Help Screen nor the Tutorial was ever touched.

How is this Possible?

```
VStack {
    if gameMode == .SURVIVAL {
        Text("Place 30 blocks without dropping more than 5 blocks! Score decays over time! Bonus
    } else if gameMode == .RACETIME {
        Text("Place 30 blocks in 60s (singleplayer) / 30s (multiplayer)!")
    } else if gameMode == .SANDBOX {
        Text("A chill, infinite game mode for you to do anything you want. Get creative!")
    } else if gameMode == .TALLENOUGH {
        Text("Build your tower as high as you can... Until the 10th (singleplayer) / 5th (multip
    }
}
```

Example of how Tutorial for Different Game Modes can be Created in Help Screen

If we use the traditional naive method of specifying what text to display in the tutorial (like shown in the above image) or in the GameEndScreen, this would not be possible as the classes that uses Game Mode have to be modified to extend the if-else or switch-case statements with new cases. Hence we look at the alternative, more extensible and Open-Closed Principle compatible method that we used:

FIrst we see what the 2 new lines in Constants are for, to look at how the enum and dictionary combination works.

```
static let gameModeTypeToClass: [String: GameMode.Type] = [GameModeTypes.SURVIVAL.rawValue: SurvivalGameMode.self,
                                                           GameModeTypes.RACECLOCK.rawValue: RaceTimeGameMode.self,
                                                           GameModeTypes.SANDBOX.rawValue: SandboxGameMode.self,
                                                           GameModeTypes.TALLENOUGH.rawValue: TallEnoughGameMode.self]

enum GameModeTypes: String, Equatable, CaseIterable {
    case SURVIVAL = "Survival"
    case SANDBOX = "Sandbox"
    case RACECLOCK = "Race the Clock"
    case TALLENOUGH = "Scale the Heights"
}

static func getGameModeType(from: GameModeTypes) -> GameMode.Type? {
    gameModeTypeToClass[from.rawValue]
}
```

Location of the 2 lines to modify in Constants file when adding new Game Mode

Instead of using if-else clauses or switch-case clauses throughout the code to check for different types of game modes (such as checking for game mode type in Tutorial then displaying the appropriate description), we use a combination of enum and dictionary instead. This makes it such that the user of game mode does not need to modify existing functions and classes to add more cases to the if-else or switch-case statements, and can just use the function getGameModeType no matter what game modes there are implemented.

Now we look at how the Help Screen made use of this enum and dictionary combination to retrieve the description of the new game mode to display in the Tutorial without any modifications to Help Screen.

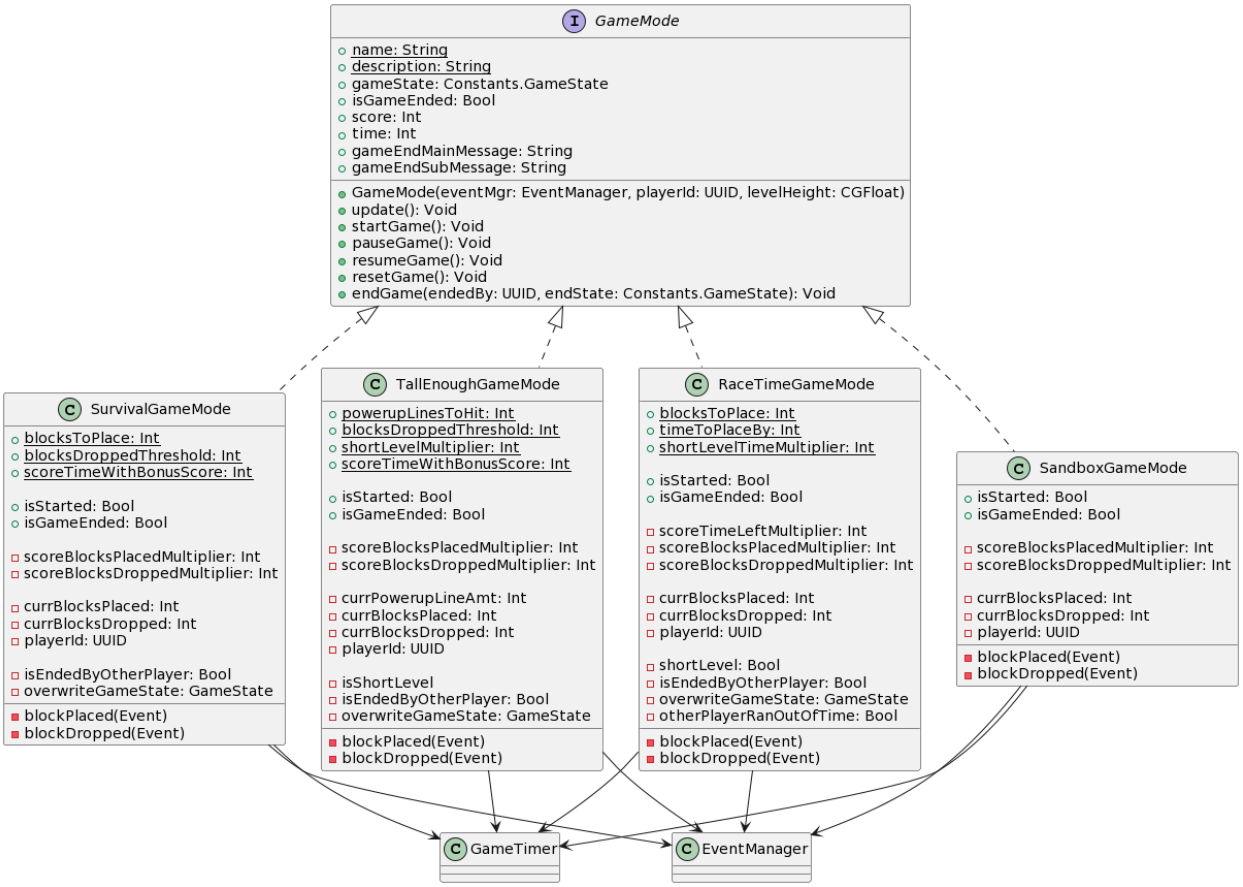
```
private func drawGameModeDescriptions() -> AnyView {
    AnyView(
        VStack(alignment: .leading) {

            ForEach(Constants.GameModeTypes.allCases, id: \.self) { value in

                HStack {
                    Text(value.rawValue + ": ")
                        .modifier(CategoryText())

                    Text(Constants.getGameModeType(from: value)?.description ??
                        Constants.instructionsDefaultGamemodeText)
                        .modifier(GameplayBodyText(fontSize: 30))
                }
                .padding(.all, 10)
            }

        }
        .background(
            RoundedRectangle(cornerRadius: 10, style: .continuous).fill(Color.init(red: 0.969,
                green: 0.933, blue: 0.855)))
    )
}
```

Help Screen code that displays descriptions of all game modes in Tutorial

The Help Screen iterates through all GameModeTypes, which contains the newly added Scale the Heights game mode. It then retrieves the actual classes for all available game modes using the Constants.getGameModeType function, which allows the Help Screen to access properties and functions of all available game mode classes, including new ones, even without any changes to it.

Module Structure

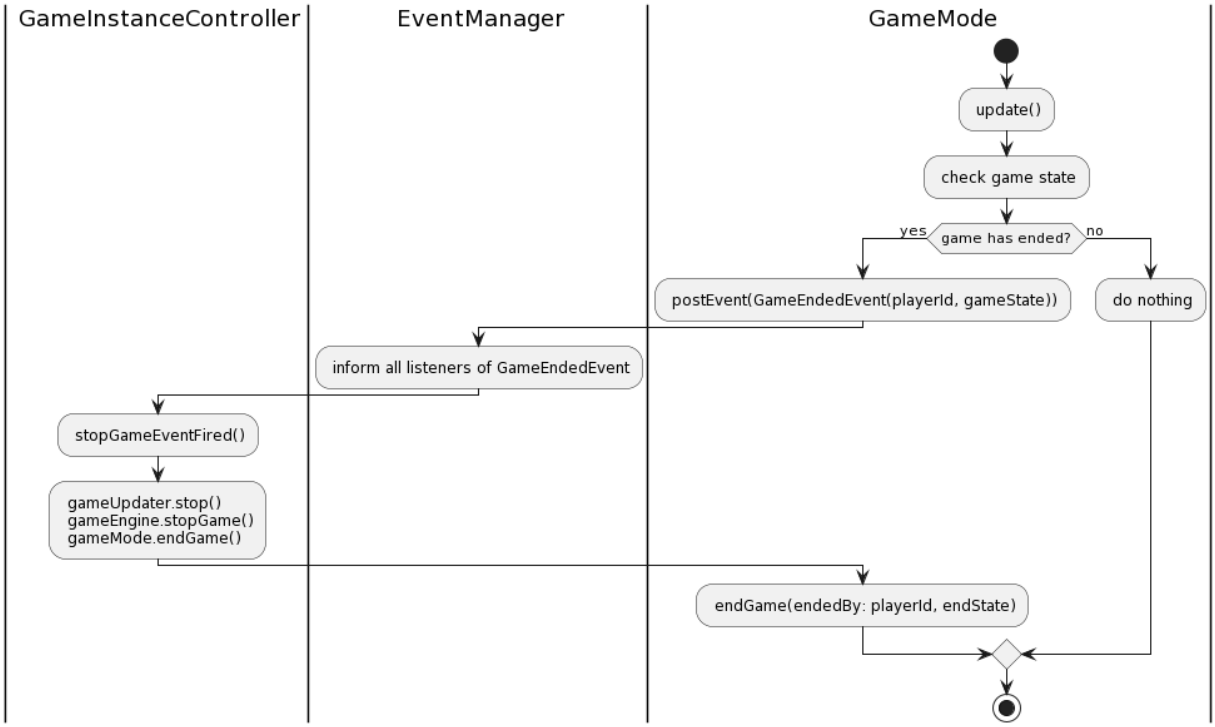
This is the full class diagram including private variables for our current implementation of game mode as described in **Option 3** in the previous section.



Runtime Structure

Winning/Losing a game

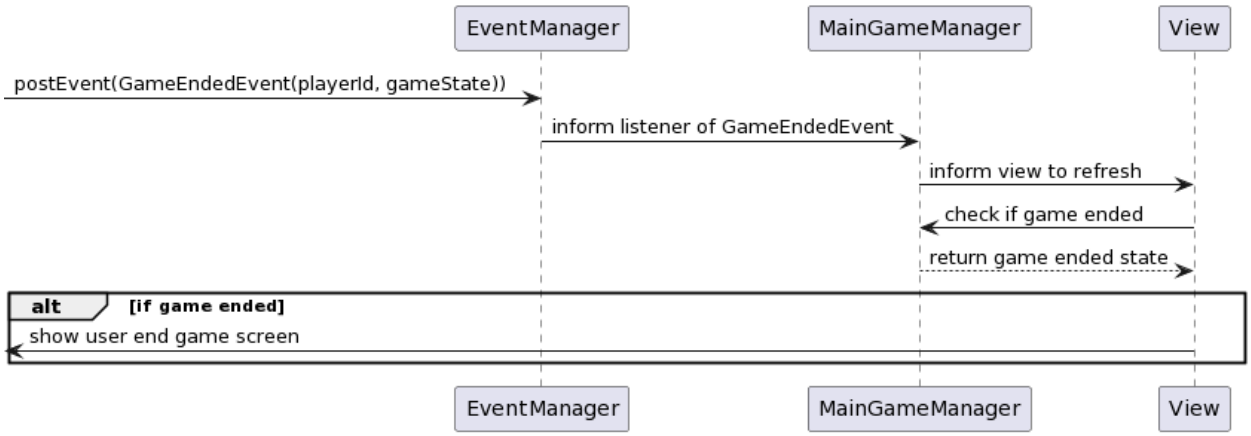
When GameEngine calls update(), which occurs every cycle of the game loop, the following occurs:



This activity diagram shows how during each run of the game loop, the game mode checks the game state to find out whether the game has ended. If the game had ended, GameMode will post the GameEndedEvent to trigger a series of events.

One of the listeners of GameEndedEvent is the GameInstanceController. After knowing that the game has ended, the GameInstanceController will inform the GameUpdater and GameEngine that the game had should be stopped, before telling GameMode to end the game.

Another listener of GameEndedEvent:

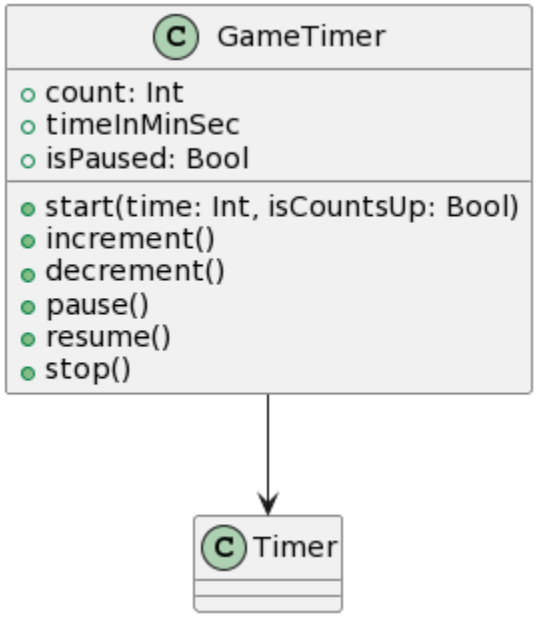


This sequence diagram shows how the View knows that the end game screen should be shown.

Another listener of the GameEndedEvent is the MainGameManager. After knowing that the game has ended, the MainGameManager, an ObservableObject, would post to its subscribers, the View, that the view should refresh itself. As part of the refresh process, the View would check whether the game had ended, and if so then display the end game screen to the user.

Game Timer

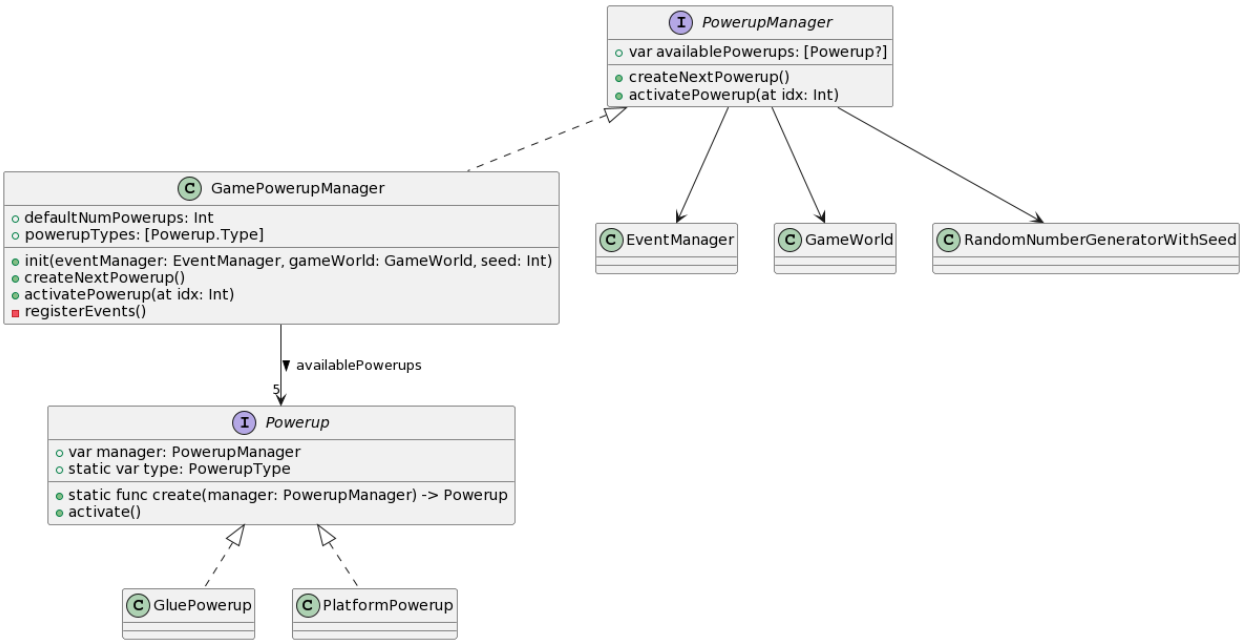
GameTimer is used by GameMode for timekeeping. Its structure is as follows:



Under the hood, GameTimer uses Apple's built in Timer to carry out timekeeping. It serves as an adapter class to the Timer class. Different game modes can set the Game Timer to have different behavior, such as being able to count up or down, or the speed of the timer (Not implemented as no game mode uses it now but can be easily added by changing the increment/decrement interval).

Powerups

Module Structure



PowerupManager is a protocol used by GameWorld to handle the creation, activation and managing of the state of power ups in the game.

GamePowerupManager uses a RandomNumberGeneratorWithSeed to generate a pseudo-random sequence of power ups. The reason for this is discussed in the Design Considerations section, under [Deciding the next powerup](#).

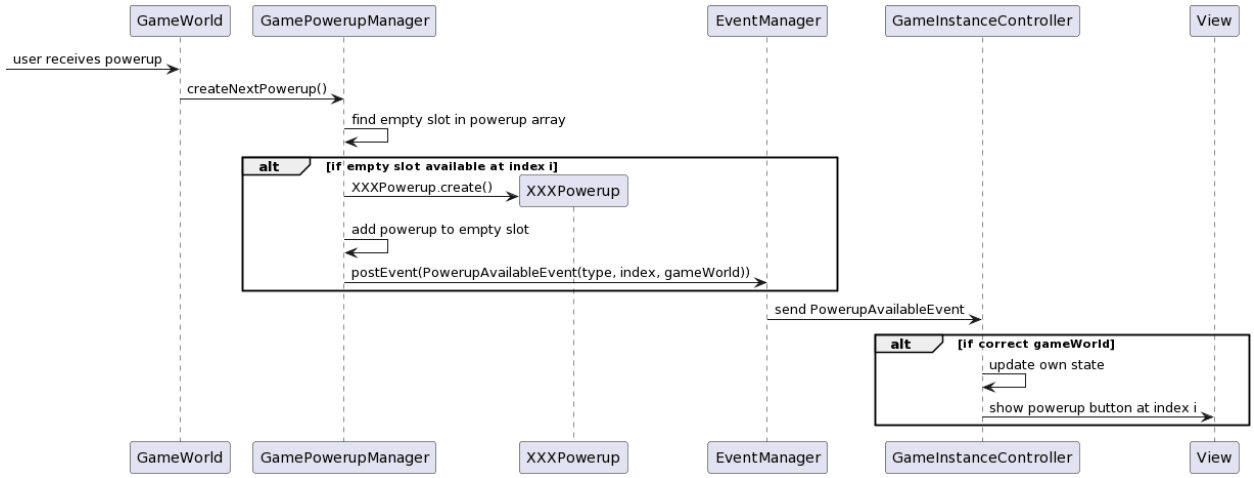
PowerupManager also stores a reference to GameWorld, in order for the activate() function in Powerup to alter the state of the GameWorld.

Runtime Structure

The user can have up to 5 power ups at any given moment, and can choose which power up to activate. The index of the chosen power up will be denoted as i in this section.

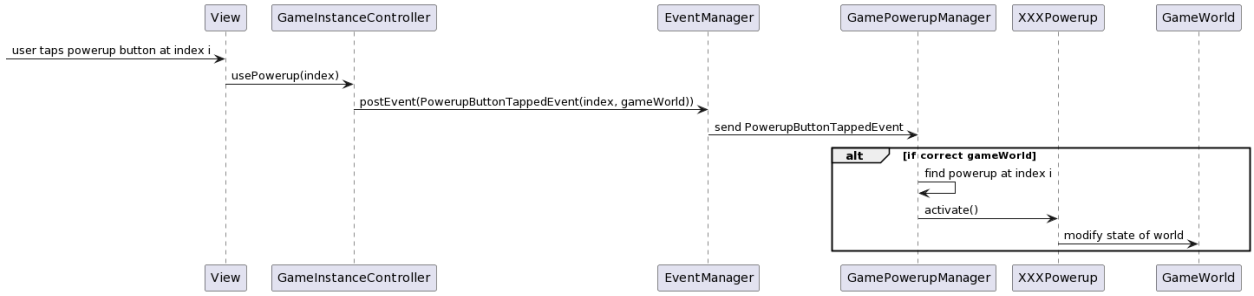
Obtaining a power up

The following diagram illustrates the entire sequence of a user obtaining a power up when the tower crosses the power up line:



Using a power up

The following diagram illustrates the sequence of using a power up at index i:



In the activate() function, the powerup will modify the state of the world based on the supposed effect of the powerup, e.g. add a new platform to the world.

Design Considerations

The implementation of the power up system is yet another interesting design consideration. There are a few points of consideration:

1. How to decide the next power up?
2. How to activate a power up?
3. How to carry out the effects of the power up?
4. How to transfer the state of the power ups from model to view?

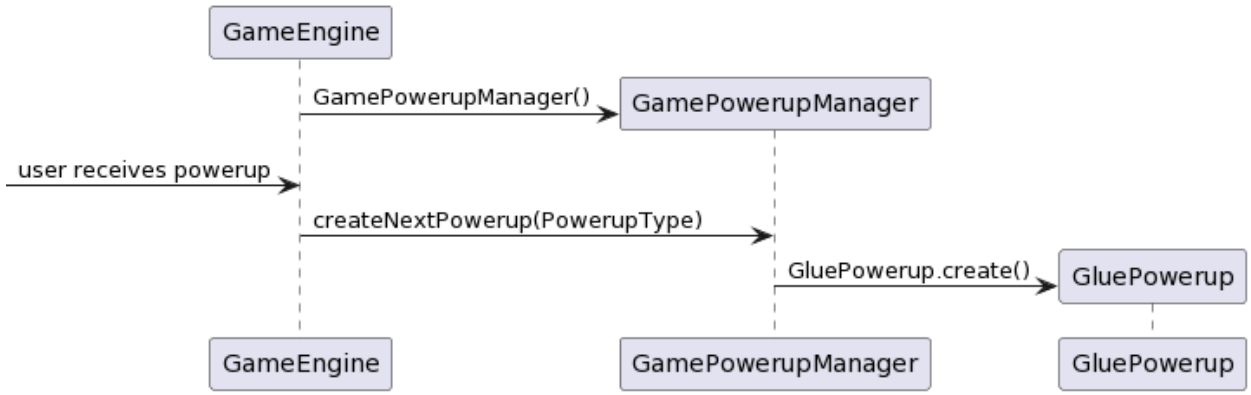
The considerations for these issues will be discussed in this section.

Deciding the next power up

When the user obtains a power up (by reaching the power up line), the power up given to the user should be (seemingly) random. This is easily achievable with a random number generator.

However, the issue arises when we consider multiplayer. We would want all players to obtain the same power up sequence in a multiplayer game (in order to achieve fairness). As such, two options arise as to who should be deciding the next power up.

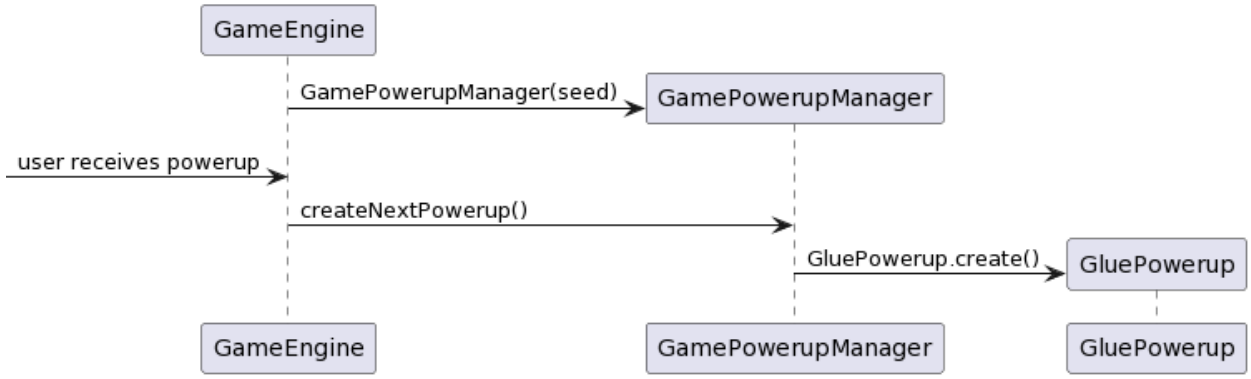
Option 1: GameEngine decides what power up is next



If the GameEngine is the one choosing the next power up type, then it would use either a RandomNumberGenerator or RandomNumberGeneratorWithSeed (depending on whether the sequence should be pseudo-random).

In the event of multiplayer, one of the GameEngines could serve as a master which would decide the power up sequence for the rest of the other GameEngines.

Option 2: Powerup Manager decides the next power up.



In this implementation, the Powerup Manager would be the one deciding the sequence of the power ups.

To achieve the same sequence across players, the Power Up Manager can use a seed, which it would use to generate a pseudo-random sequence. This would ensure that the sequence is consistent across all players.

Choice
We chose **Option 2**.

This option ensures Separation of Concerns, such that the GameEngine need not concern itself with the power up sequence in addition to its other responsibilities. In addition, it makes the implementation of multiplayer simpler, as the Game Engines would not need to communicate with each other, and just run an independent instance.

However, the limitation is that the GameEngine has no control over the power up sequence (e.g. changing it halfway through the game), since the sequence will already be decided when the game begins (via the pseudo-random number generator). We deem this limitation insignificant, as in the first place, it is more important that all players get the same sequence (for fairness of the game).

Creation of Powerups

In the powerup creation sequence, the issue arises on how to create the new instance of Powerup. We could apply a creational pattern such as the Factory pattern in this instance.

Option 1: Using a switch statement

We could do something like this:

```
func createPowerup(type: PowerupType) -> Powerup? {
    switch type {
    case .glue:
        return GluePowerup()
    case .platform:
        return PlatformPowerup()
    default:
        return nil
    }
}
```

However, ideally we wouldn't want to use a switch statement, as each time we add a power up we would have to add another case in this switch statement. Ideally, we would want minimal changes in the GamePowerupManager file whenever we add a new powerup, following OCP.

Option 2: Array of powerup types

Alternatively, we could do this:

```
static let powerupTypes: [Powerup.Type] = [PlatformPowerup.self, GluePowerup.self]
```

Whenever a new powerup needs to be created, we can use the random number generator to get the index of the powerup type to create, and then just call a static create() on the type as such:

```
func createNextPowerup() {
    let next = rng.next()
    let idx = Int(next) % GamePowerupManager.powerupTypes.count
    let type = GamePowerupManager.powerupTypes[idx]

    if let index = availablePowerups.firstIndex(where: { $0 == nil }) {
        var nextPowerup = type.create(manager: self)
        availablePowerups[index] = nextPowerup
        eventManager.postEvent(PowerupAvailableEvent(type: type, idx: index, for: gameWorld))
    }
}
```

```
static func create(manager: PowerupManager) -> Powerup
```

static create() in Powerup.swift

Then, whenever we want to add a new power up, we can just add the type to the powerupTypes array, resulting in minimal changes, conforming to OCP.

Choice

Based on the discussion above, to more closely follow OCP, we chose **Option 2**.

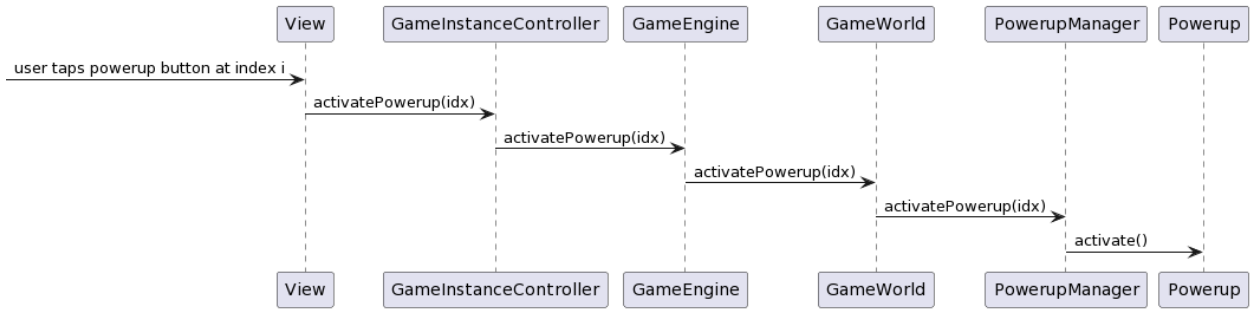
Activating power up

The user can have up to 5 power ups at any point in time, and can select which power up they wish to use. While designing the sequence for activating a power up, we need to consider a few things:

- How do we activate the power up for the player that activated it only?
- How do we activate the power up that was chosen by the user?

Option 1: Series of function calls

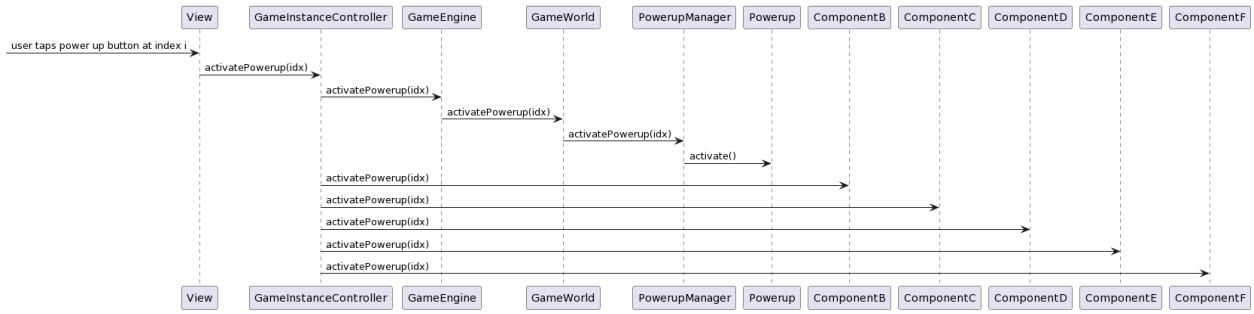
The most straightforward way would be a series of function calls as shown below:



When the user taps the power up button at index i, this series of function calls would occur just to activate a powerup. There are a few issues with this:

First, this long chain of function calls is difficult to follow, and adds additional responsibilities onto the GameEngine and the GameWorld in having to “pass” this function call to their child. Ideally, we only involve the GameInstanceController, the PowerupManager and the Powerup itself in the activation sequence.

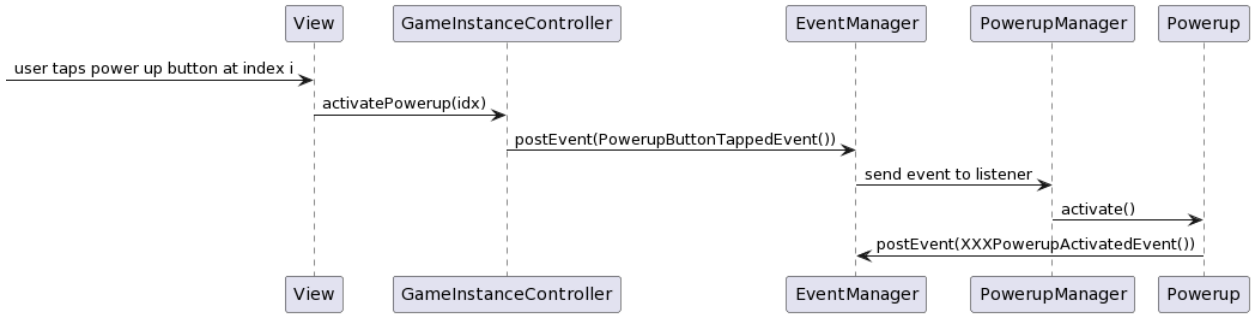
Second, if other components need to know that a powerup has been activated (e.g. a statistic tracker needs to track the number of power ups activated), it would lead to a sequence as such:



This would lead to increased complexity and unnecessary bloating of the code base. We could mitigate this by using an Observer pattern, but this would lead to the same problems as discussed in [Event Manager](#).

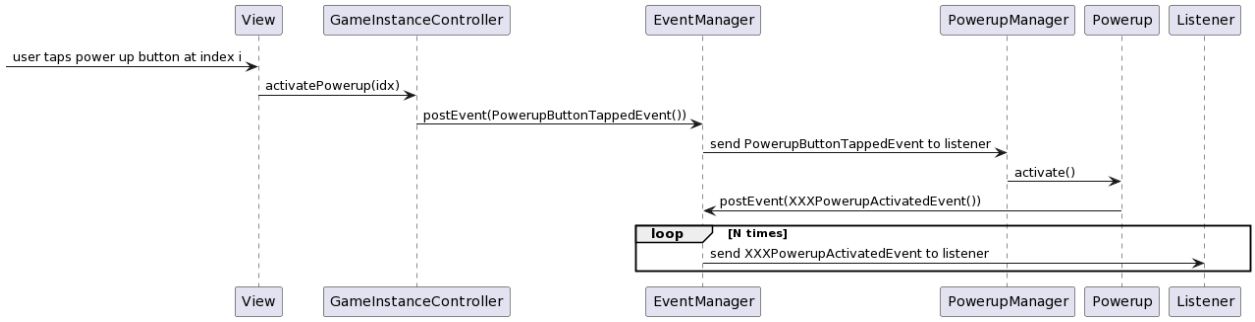
Option 2: Using Events

A better solution would be to use an event system to propagate the event to the components which are interested.



When activatePowerup() is called on GameInstanceController, instead of a series of function calls, GameInstanceController would post the PowerupButtonTapped to the EventManager. The EventManager would handle the publishing of this event. The Powerups activate() method would then post another event, XXXPowerupActivatedEvent().

If multiple components wish to know about the event:



The EventManager would simply post to all its listeners that the event has occurred.

This also allows components to only listen to specific power up activations, e.g. a component could listen to only GluePowerupActivatedEvent.

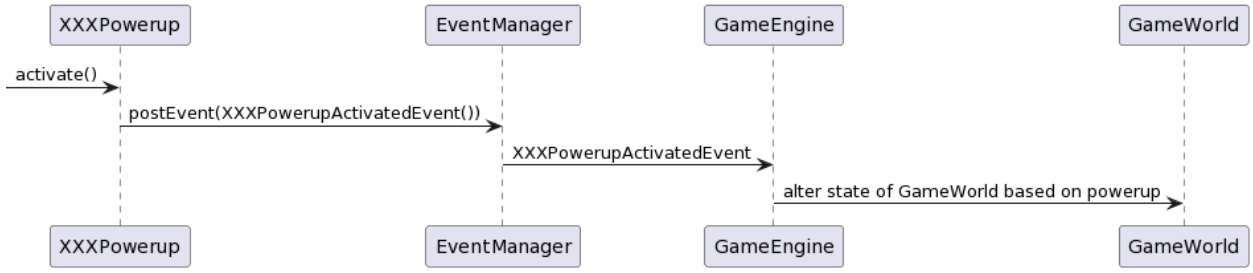
Choice

Due to the limitations of Option 1 and the flexibility of Option 2, we chose **Option 2**.

Carrying out effects of powerups

When the power up is activated, it would have effects on the GameWorld. How these effects are applied onto the GameWorld would be the focus of the discussion in this section.

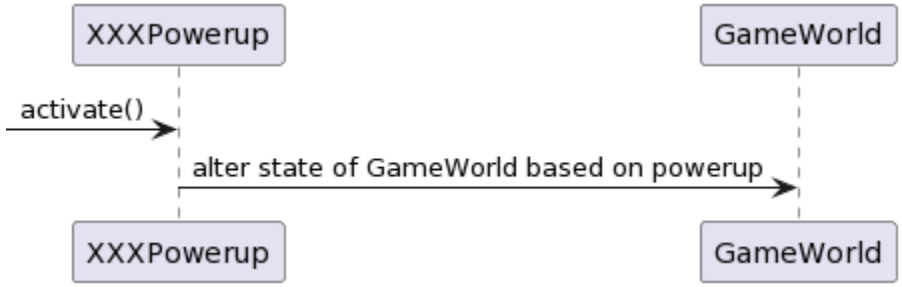
Option 1: **GameEngine listens to events and carry out effects**



When a power up is activated, GameEngine is informed and calls functions of GameWorld to alter its state (e.g. adding a new platform).

The issue with this is that it violates Open Closed Principle (OCP). If we consider adding a new powerup X, we would have to register GameEngine with a new event, XPowerupActivatedEvent, which violates OCP as we should not have to modify GameEngine to add a new powerup.

Option 2: Powerup alters GameWorld directly



When we want to add a new powerup X, we just need to create a XPowerup class and specify what state to change in GameWorld in XPowerup's activate() function. This follows OCP, as the system is open to extension, without having to modify anything else.

Granted, if certain functions that are needed to alter the GameWorld state are not present, we would need to add those functions as we do not have a choice. If the current methods exposed by GameWorld are sufficient, we do not have to alter GameWorld as well.

For a more detailed explanation on how to add a new powerup, refer to [Adding a new powerup](#).

Choice

Based on the discussion on OCP above, we chose **Option 2**.

Informing View powerup availability

This discussion is similar to the one in Activating power up, where we discussed how to propagate an event from the View to the Model. Similarly, the options are:

1. A series of function calls from Model all the way to GameInstanceController, to update some state in GameInstanceController.
2. GameInstanceController would listen to the event that a powerup is available, and change its own state.

View would then see the change in the state in GameInstanceController and render a button.

Choice

For reasons similar to the choice in Activating powerup, we chose **Option 2**.

Adding a new powerup

The design of the Powerup Manager takes into account OCP - adding a new powerup should be easy and should not involve modification of other unrelated files.

We operate under the assumption that any methods to modify the state of the GameWorld needed by XXXPowerup are already available (otherwise, there is no choice, we need to modify GameWorld in order to allow the Powerup to modify the state of the GameWorld in the way it needs to).

Below are the steps that a developer could take to add a new powerup:

1. Create the XXXPowerup class which extends from the Powerup protocol.
2. Add the type of the powerup to GamePowerupManager or the current PowerupManager being used:

```
static let powerupTypes: [Powerup.Type] = [PlatformPowerup.self, GluePowerup.self, XXXPowerup.self]
```

3. Specify the logic that will modify the GameWorld inside the activate() function (this function is needed to conform to Powerup protocol):

```
func activate() {
    // add logic here to apply power up effect to GameWorld
}
```

Below is an example, using PlatformPowerup's activate() function:

```
func activate() {
    guard let newPlatform = createPowerupPlatform() else { return }
    gameWorld?.addObject(object: newPlatform)
    eventManager?.postEvent(PlatformPowerupActivatedEvent())
}
```

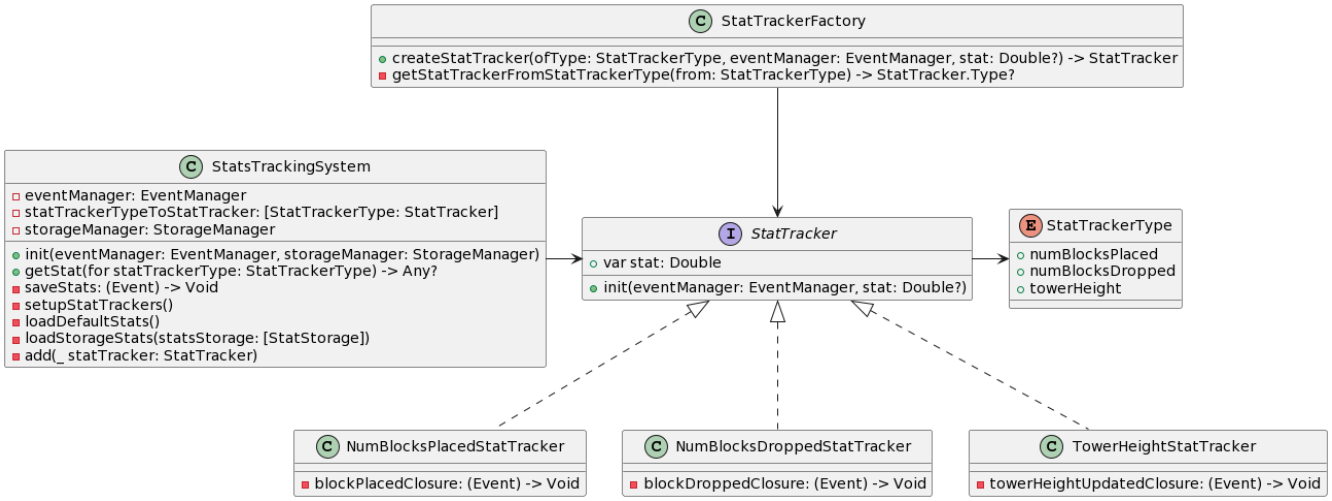
4. [OPTIONAL] Create the powerup activated event class, i.e. XXXPowerupActivatedEvent. This is only required if there are some other components outside the model, e.g. SoundSystem, which needs to know when the XXXPowerup is activated (for example, to play a certain sound).

These 4 steps are all that is required to add a new power up. We can see that only the GamePowerupManager file is modified during the addition of the XXXPowerup.

Stats Tracking System

The StatsTrackingSystem is created with the purpose of tracking achievements. In the following section on the AchievementSystem, the StatsTracking System will be used as a data source. However, the StatsTrackingSystem is capable of being used in other contexts in future, such as high score tracking in the multiplayer setting.

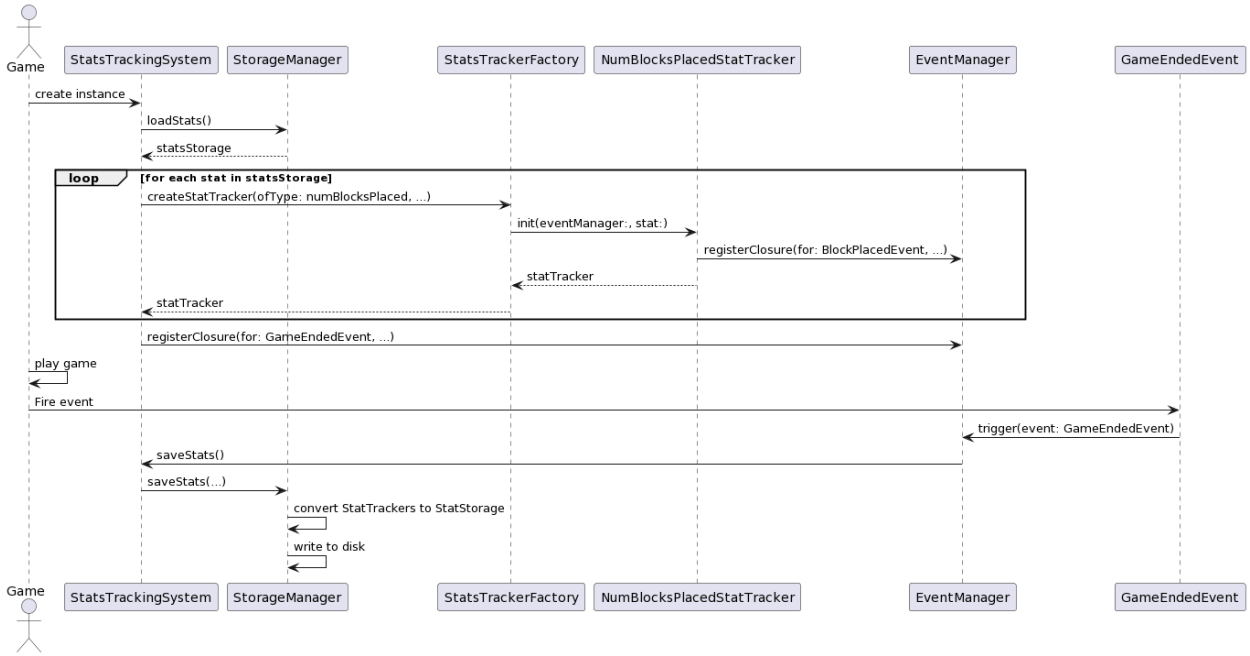
Module Structure



The StatTrackingSystem maintains a collection of StatTrackers.

For the creation of StatTrackers, the Factory pattern is used. This abstracts away the logic for the creation of StatTrackers to the Factory class, and allows new StatTrackers to easily be created and used by the StatTrackingSystem.

Runtime Structure



Upon instantiation, StatTrackingSystem either:

1. Loads in stats available in the StorageManager, or
2. Loads in a set of predefined StatTrackers in their initial state.

When the GameEndedEvent is fired, StatsTrackingSystem saves all stats to the StorageManager.

Design Considerations

Option 1: StatsTrackers hold a stat of `Any` type

This will allow for the case where users want to track new stats of various types for example CGVector or any other type. However, for the actual implementation, this would mean that there is a need to downcast the stat from `Any` type to the required type. While this can be implemented in practice it is extremely unfeasible, especially when considering the conversion of data of `Any` type to be stored in the StorageManager since this will require a case-by-case casting. Also, the conversion back from the stored data type to the data type in the individual StatTrackers needs to be done in a similar fashion as well.

There can be a workaround where the encode and decode function is dynamically called. This can be done by having a parent class and force it to act like an abstract class in Java, i.e. create empty encode() and decode() functions where they throw fatal error. Next, have the child classes (the individual StatTrackers) have an extension that overrides the encode() and decode() function by force casting the data types, as they would know the stat data type during runtime.

However this makes things very messy as for every StatTracker class, a new class or an extension has to be created. A solution for this is to possibly have implemented protocols (via extensions) for each data type and let the StatTracker classes conform to it.

An issue with this entire roundabout implementation is that protocols in swift are often limited where it has to adhere to specific requirements and these might introduce more complexity. One example is that the error “xxx protocol cannot conform to Codable” even when it does when attempting to do so.

Option 2: StatTrackers hold a stat of `Double` type

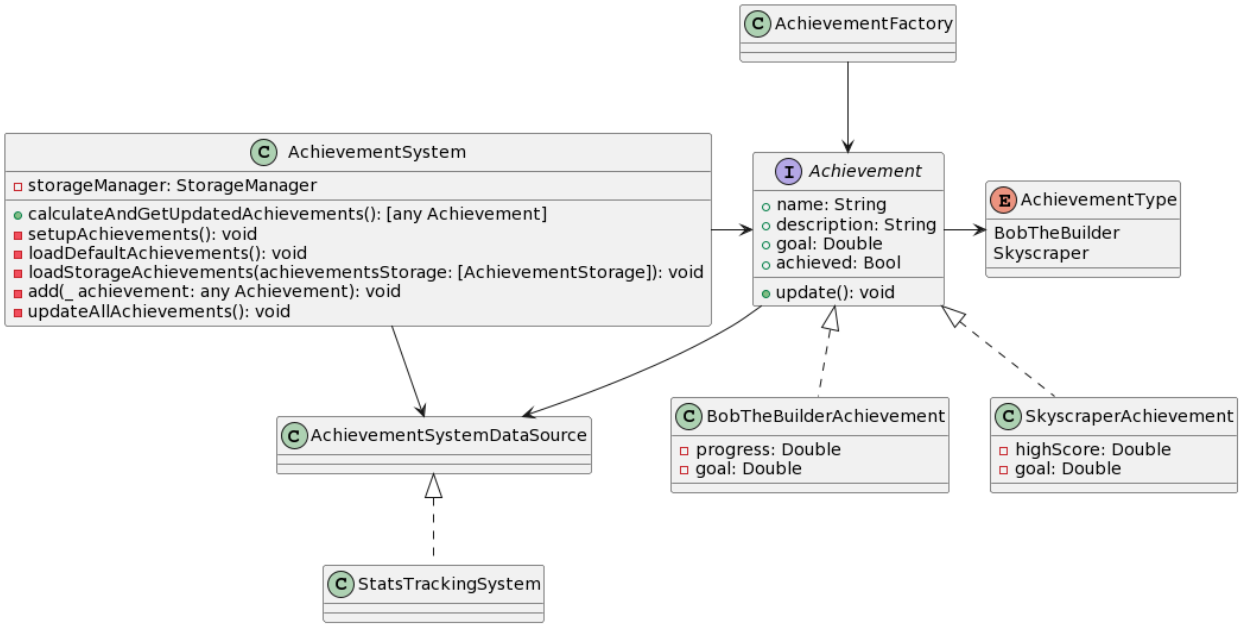
The most obvious drawback to this is that all stats tracked must be numerical, and it limits extensibility in terms of tracking other types of stats. However, after putting some thought into a StatTrackingSystem’s use cases and considering most Stats Tracking Systems in other games, we can see there is almost never a case where stats tracked are non-numerical. Therefore, it is still reasonable to select this option despite the drawback.

The advantage of this option is that there is no longer a need to perform any form of typecasting nor is there a need to maintain some form of mapping containing information about the different types of stats tracked by different StatTrackers. Additionally, storing with the StorageManager becomes a much simpler task.

Therefore, **Option 2** was selected for the actual implementation.

Achievements System

Module Structure

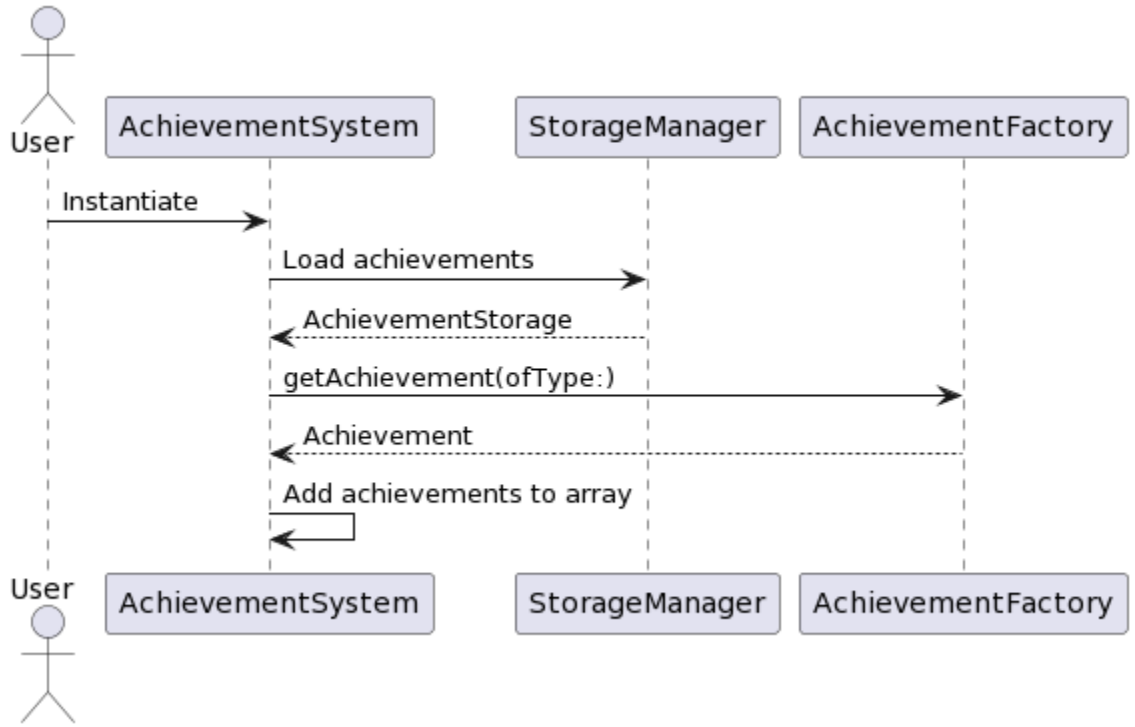


Similar to the StatTrackingSystem, the AchievmentSystem maintains a collection of Achievements.

For the same reason as StatTrackers, the factory pattern is also used for the creation of Achievements.

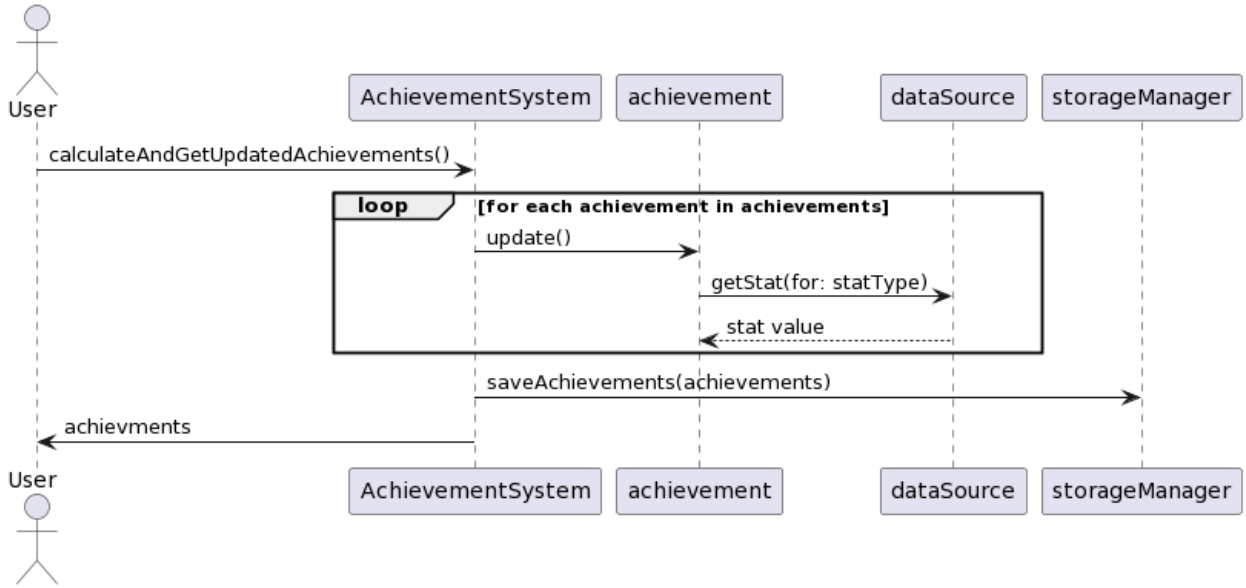
Each Achievement has a reference to an AchievementSystemDataSource, which the StatsTrackingSystem conforms to. A use case example is described below.

Runtime Structure



Upon instantiation, the AchievementSystem either:

1. Loads in achievements available in the StorageManager, or
2. Loads in a set of predefined Achievements in their default state (not achieved).



Whenever Achievements need to be accessed (e.g. when the Achievements screen is brought up), a query is sent to the AchievementSystem. The 'calculateAndGetUpdatedAchievements()' method loops through all achievements in the AchievementSystem and calls their 'update()' methods. This triggers the Achievement to get the latest stats from the StatTrackingSystem and check if the Achievement's conditions have been met. The most updated Achievements are also stored by the StorageManager and then returned.

Design Considerations

Option 1: Each Achievement has its own XXXAchievementDataSource, which StatsTrackingSystem has to conform to

The benefit of this option is the clarity provided by each XXXAchievementDataSource protocol. The protocol would only contain the values required by the respective Achievement, thus ensuring that when implementing the Achievement's logic, the user would never accidentally use the wrong type of data.

However, the drawback of this option is that each time a new Achievement is added, the StatsTrackingSystem would need to be extended with a new protocol and new logic to obtain the data required by the protocol. This violates OCP and makes the process of adding new Achievements more complicated than necessary.

Option 2: The StatsTrackingSystem conforms to a single AchievementSystemDataSource

The advantage of this option over **Option 1** is that since each Achievement has the ability to access all the data in StatsTrackingSystem, there would be no need to extend the StatsTrackingSystem each time we add a new Achievement. This allows all the logic for a certain Achievement to be contained in the Achievement class, thus making the process of adding new Achievements as simple as creating the Achievement class and adding it to the AchievementSystem.

This option was selected as it properly follows OCP and allows for easy extension.

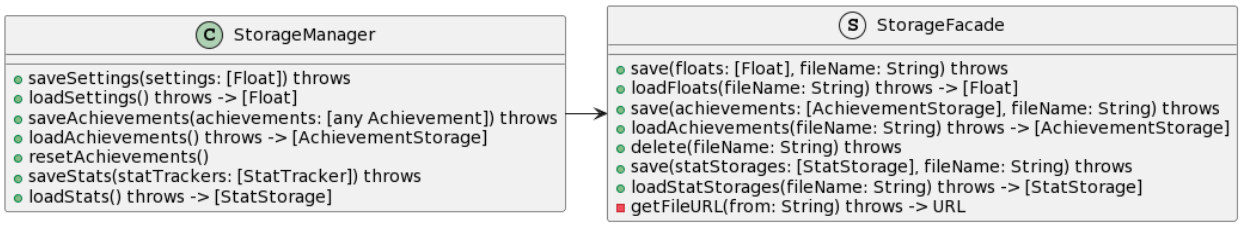
Storage

The storage class handles storing and loading data on the device. The implementation uses property lists and handles classes that conform to the Codable protocol. Currently persistence is only needed for the sound settings, stats and achievements.

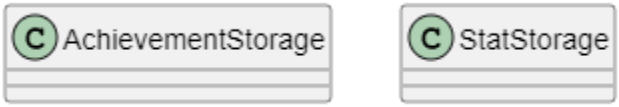
Module Structure

There exist 2 main files, StorageManager and StorageFacade.

StorageFacade holds the actual implementation of how the data is being stored. In our case, we used the **Codable protocol.** In the event that the storing method is changed to CoreData, SQL etc, only StorageFacade has to be changed.



As we used the Codable protocol, we created additional classes that conform to Codable, and is almost identical to the original classes as shown below, i.e. AchievementStorage and StatStorage.



StorageManager is responsible for saving and loading existing data such as settings, achievements and stats by calling functions from StorageFacade. StorageManager is also the class responsible for converting the original class to the storage version of it and vice versa. This ensures that other packages and the original files do not need to know about how the storage works.

Design Considerations

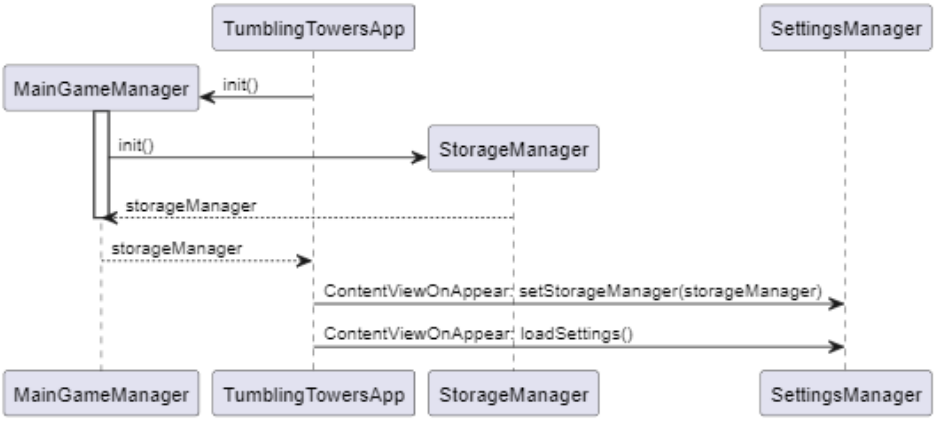
Option 1: StorageManager as a singleton
Implementing StorageManager as a singleton adds convenience when developing as well as this reduces the need to pass the StorageManager from parent component to child component multiple times (similar to prop drilling in React) in the codebase.

However, as the StorageManager class has a state that can be modified by other classes, exposing StorageManager as a singleton would risk a case where a future developer unknowingly and wrongly modifies StorageManager, thus affecting all other classes that use it.

Option 2: Initialize 1 StorageManager object and pass it to relevant classes
We can create a single StorageManager (not a singleton instance) that is passed around. The single StorageManager would be instantiated at the main view model to be passed down to the relevant classes.

This method increases testability where stubs can be utilized. This option also has the benefit of reducing the exposure of this StorageManager instance to other classes that do not use it and hence cannot modify it and cause any unwanted side effects.

The trade off would be that this requires additional code to pass the StorageManager around the different components, increasing the complexity of the codebase, but this trade off is taken as the we want to reduce unwanted side effects.



Choice
Option 2 is chosen**.**

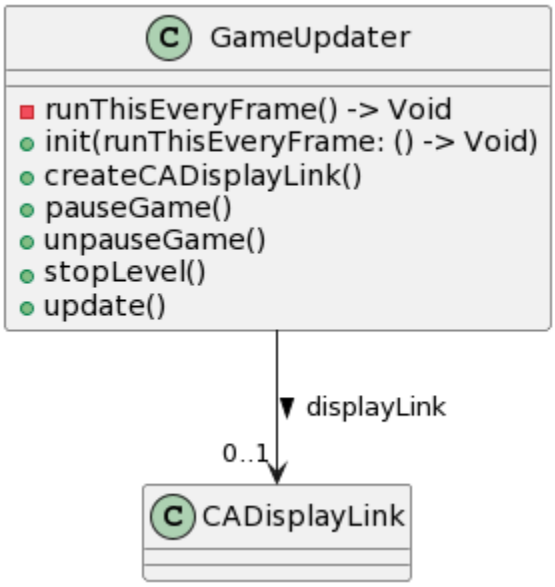
GameUpdater

The GameUpdater can be thought of as a “timer” class that runs the function given to it on every display frame, and does not hold any other reference to Tumbling Towers files.

The GameUpdater is a facade for CADisplayLink, using a CADisplayLink to synchronize the game loop to the frame rate of the device’s display. This is used by the GameInstanceController to run the game loop.

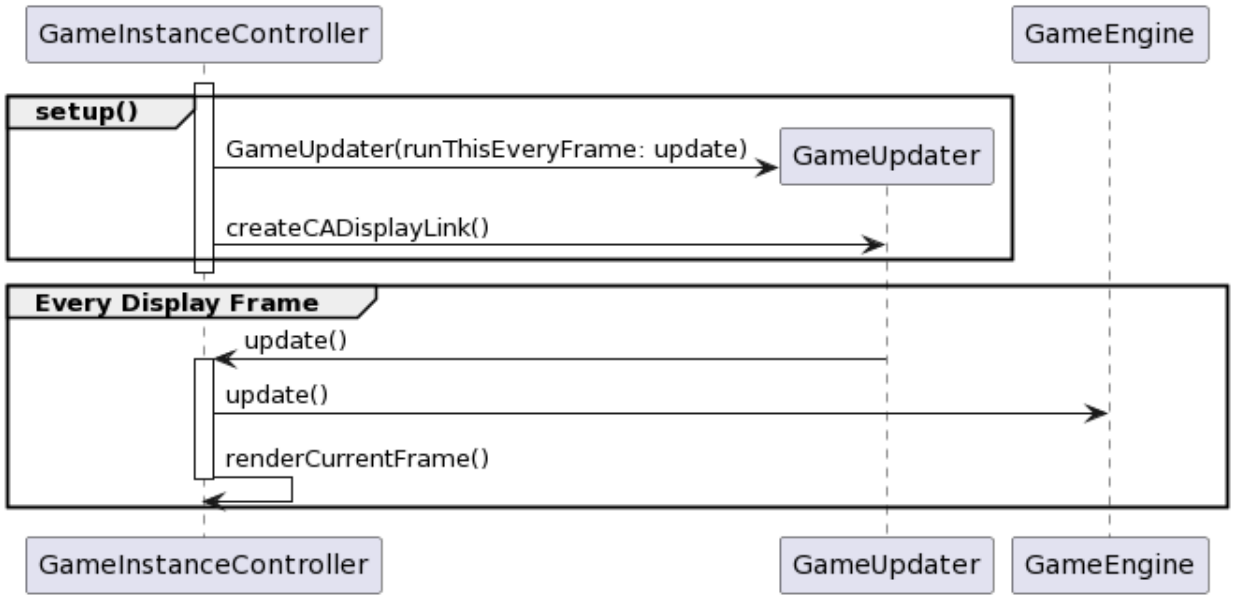
The GameUpdater contains no other dependency to TumblingTowers and hence can be used as a standalone package. It contains functions to begin, pause and stop the given function (the game loop update function in our case) from being called.

Module Structure



Runtime Structure

Game Loop - An example use of GameUpdater inside GameInstanceController



Every cycle, GameUpdater calls the update() function of GameInstanceController, which subsequently calls update() on GameEngine to update the model and renderCurrentFrame() to render the current state of the model.

Design Considerations

Option 1: Hardcoding GameUpdater to run gameInstanceController.update()
The first option is to simply specify what function should be run every display frame, in our case, gameInstanceController.update().

This method would be more straightforward and have lower implementation complexity. However, the GameUpdater would be tightly coupled to the GameInstanceController.

Option 2: Making GameUpdater standalone
Alternatively, we could pass the GameUpdater the function to run every frame, via the initializer. This allows us to run any function we want in every display frame.

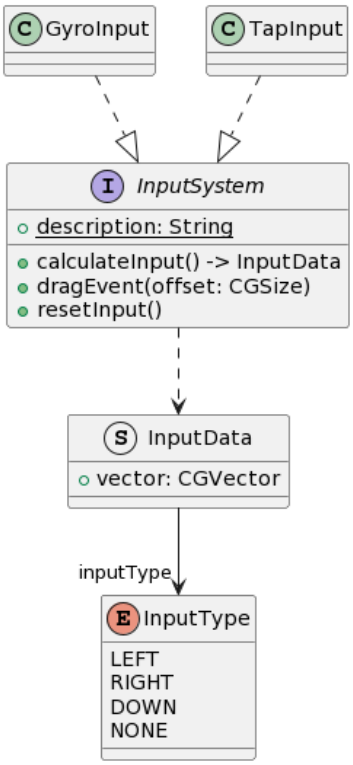
This method would be more complex, but more reusable. We could potentially have multiple GameUpdaters running at once, if needed. It also decouples the GameUpdater and the GameInstanceController, so that if we need to change the function to run every frame, we don't need to update the code inside GameUpdater.

Choice
We chose **Option 2**. We deemed the benefits of reusability to outweigh the slightly higher implementation complexity.

InputSystem

The InputSystem handles all the logic for using different methods for the user to control the game. It is used by the GameInstanceController to process all raw input from the ViewAdapter into a consistent format that can be read by TumblingTower's Models.

Module Structure



The InputSystem is a set of classes that are used to receive and handle different types of user inputs in order to control the game.

The logic for how different input types are captured (gyro, tap) are situated in their own respective classes (GyroInput, TapInput). InputSystem handles the translation of raw user input into a value that can be used by the application. GameInstanceController calculates the input using this value during each run of the game loop to know how to move the current block.

Runtime Structure

At runtime, every frame, the updateGameEngine() function is called in the GameInstanceController:

```
func updateGameEngine() {
    gameEngine.update()
    let currInput = inputSystem.calculateInput()

    gameEngine.moveCMBSideways(by: currInput.vector)
    gameEngine.moveCMBDown(by: currInput.vector)

}
```

The current input from all the user input to control the CMB (e.g. tilt device, slide finger down, etc.) will be processed by the current InputSystem and returned in the calculateInput() method. The result indicates how much to move the CMB. For example, the calculateInput() method in GyroInput:

```
func calculateInput() -> InputData {

    if let rate = motionManager.accelerometerData?.acceleration.x {
        // prioritise down input
        if inputData.inputType != .NONE {
            return inputData
        } else if rate > 0.1 {
            // If want constant rate
            return InputData(inputType: .RIGHT, vector: CGVector(dx: rate * sensitivity, dy: 0))
        } else if rate < -0.1 {
            // If want constant rate
            return InputData(inputType: .LEFT, vector: CGVector(dx: rate * sensitivity, dy: 0))
        }
    }
    return InputData.none
}
```

We can see that the raw accelerometer data is converted to InputData, which is used by the GameInstanceController.

Design Considerations

In designing the InputSystem, we wanted to ensure:

1. Easy to change input systems
2. Easy to customize input systems
3. Easy to add new input systems

Swapping Input Systems

This InputSystem allows users to easily change the input type that they want to play with. We just need to change the inputSystem attribute in MainGameManager. This can be accomplished using the changeInput() method:

```
func changeInput(to inputType: Constants.GameInputTypes) {
    inputSystem = inputType
}
```

Doing this decouples the different types of input systems from the rest of the project. This means that we do not need to modify the model to change the type of input used to control the game.

Customizing Input Systems

Each class that implements InputSystem protocol can do anything they want to retrieve the user input, and can process it however they like in order to compute the type of input to give to the game (move left, right, down).

We can individually customize the sensitivity of each input system, i.e. making GyroInput more sensitive does not affect TapInput

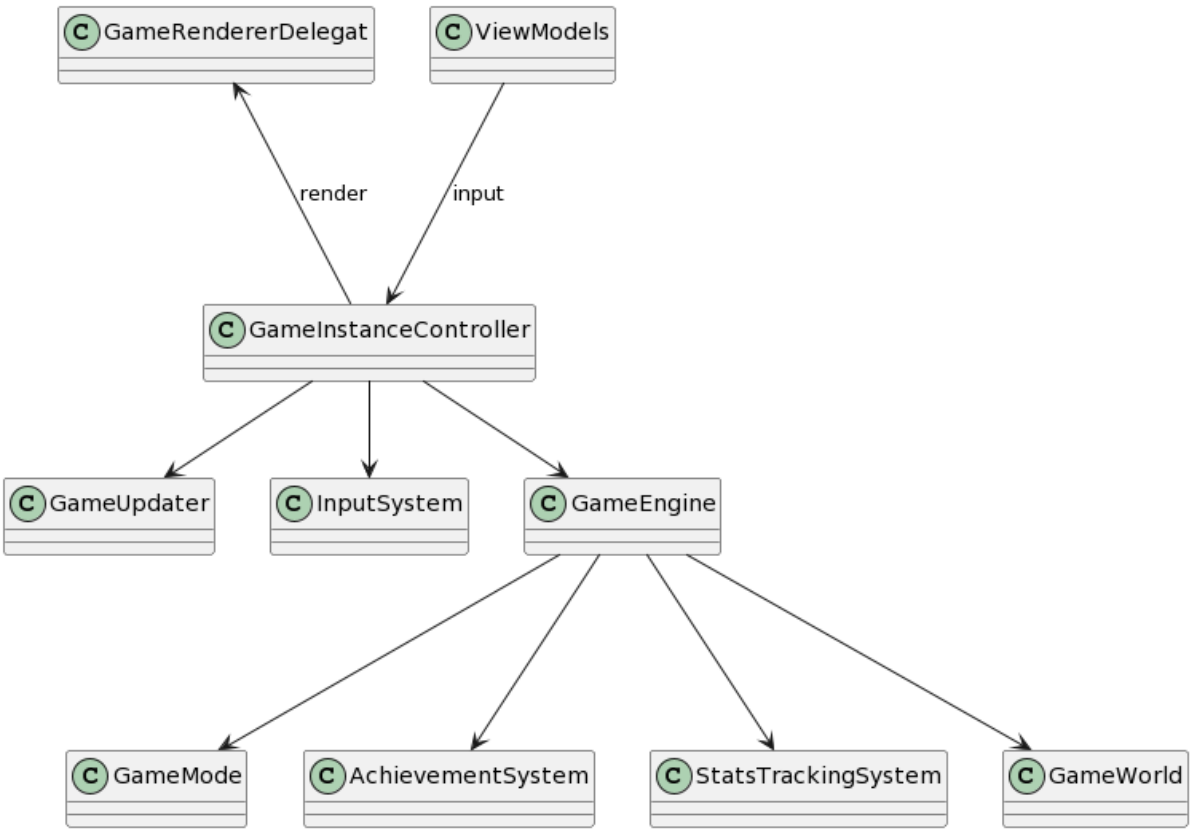
Adding More InputSystems

It is also easy to create a new type of input controller. For example, if we want to use the keyboard as input instead, we can create a KeyboardInput class which conforms to the InputSystem protocol. It imports the required libraries to receive input from the keyboard, then processes these inputs and presents the processed input to the GameInstanceController when the calculateInput() method is called.

GameInstanceController

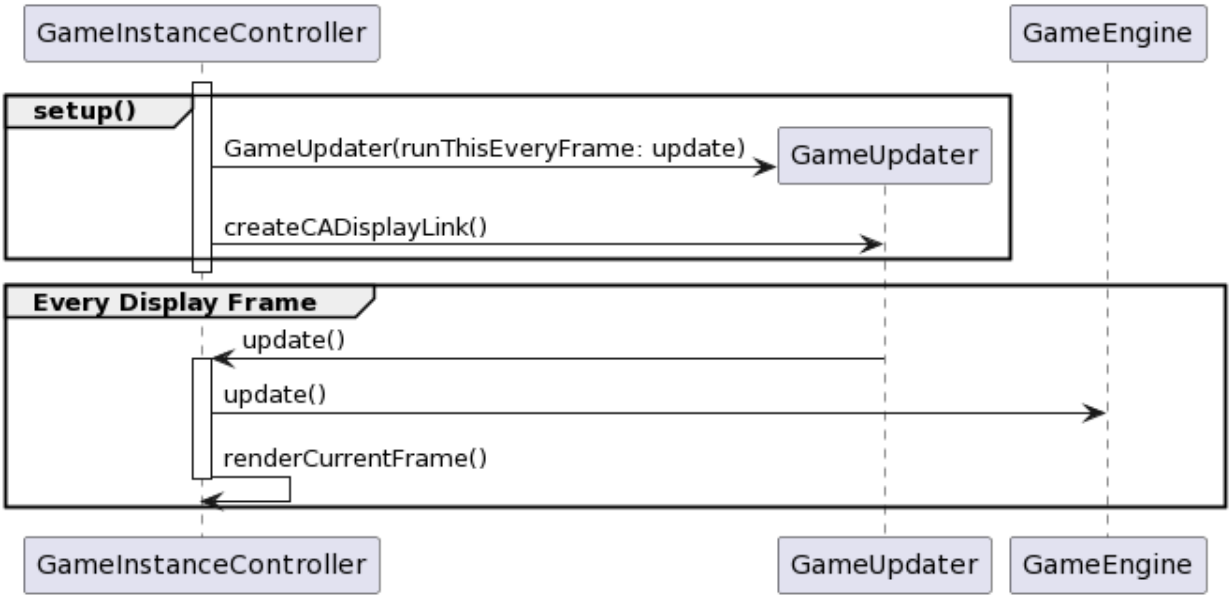
The GameInstanceController manages an instance of the game, with independent instances of the different components to allow an instance of TumblingTowers to run. The responsibility of the GameInstanceController is to perform routing between the View Model and the Model, as well as additional components like the InputSystem.

Module Structure



Runtime Structure

Game Loop



As described before, the GameInstanceController makes use of the GameUpdater to run its game loop.

During each display frame, the GameUpdater runs the update function inside GameInstanceController. The GameInstanceController would then trigger the things that needs to be done every frame, including:

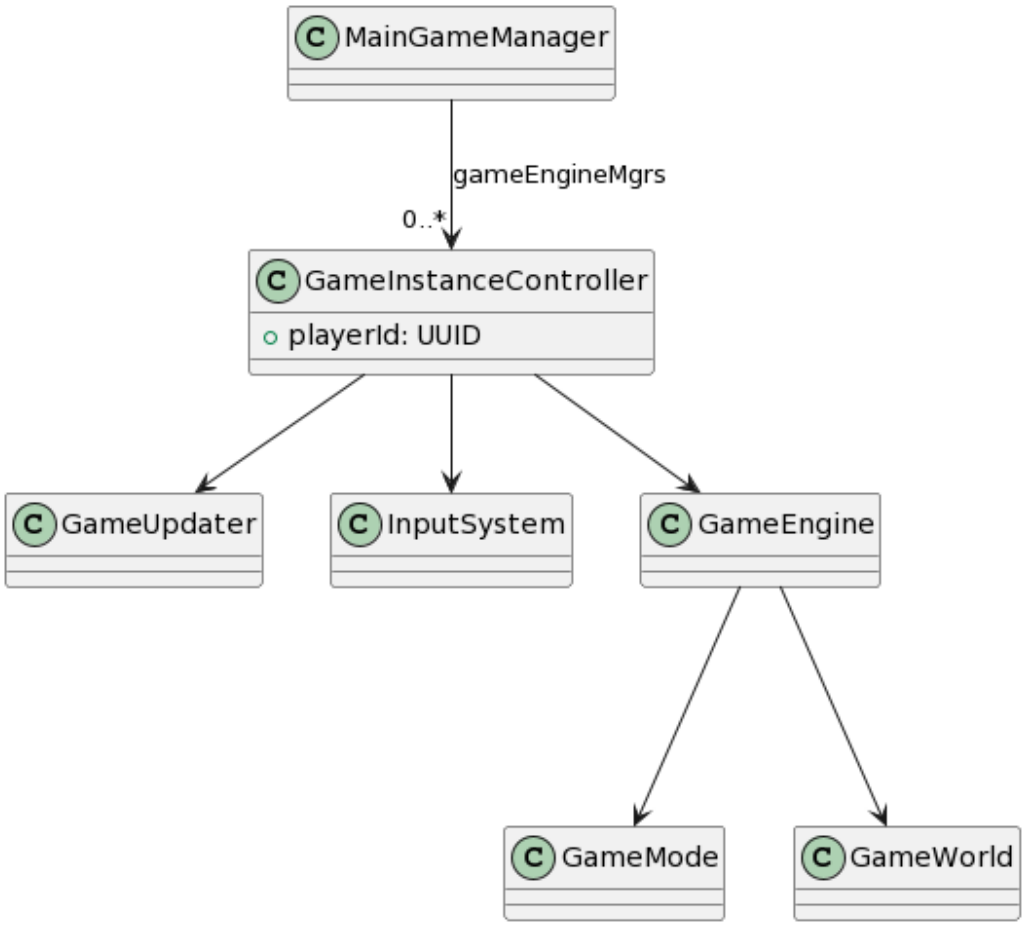
- Running the update function of GameEngine
- Rendering the Level

Multiplayer

We want to implement a Multiplayer mode in Tumbling Towers, in order to allow multiple users to play together on the same device. When designing Multiplayer, we had a few implementation goals:

- Should extend the normal Singleplayer mode so that anything new added to Singleplayer would likewise be added to multiplayer
- Should allow the developer to easily create multiplayer for any number of players

Module Structure



Our design of GameInstanceController helped us to easily implement Multiplayer for TumblingTowers. This is because the different components that interact with 1 instance of the game all have GameInstanceController as the root.

Each GameInstanceController has a playerId field, in order to differentiate between the different players of the game. The current implementation allows any number of GameInstanceControllers, and thus any

number of players to be created, as long as it fits the screen. The game can support 20 players if you want but the screen may not fit and it might not run well.

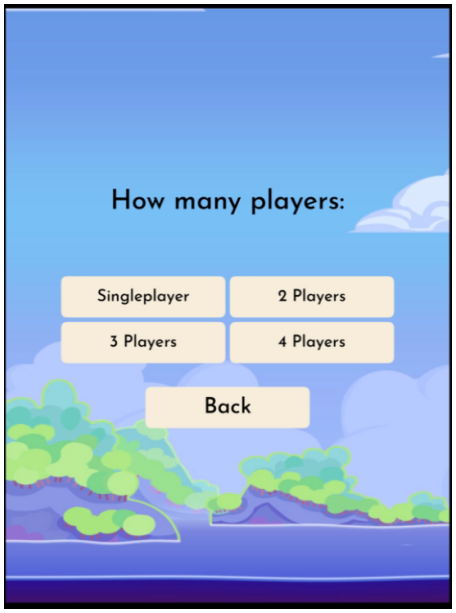
Gameplay for everyone is independent except for when the game ends. All GameModes also support multiplayer for any number of players. playerId retrieved from GameEndedEvent used to check who had won or lost the game so that the GameMode for all other players can react accordingly when GameEndedEvent is posted to the EventManager, where the game ends for any of the ongoing games.

Adding More Players

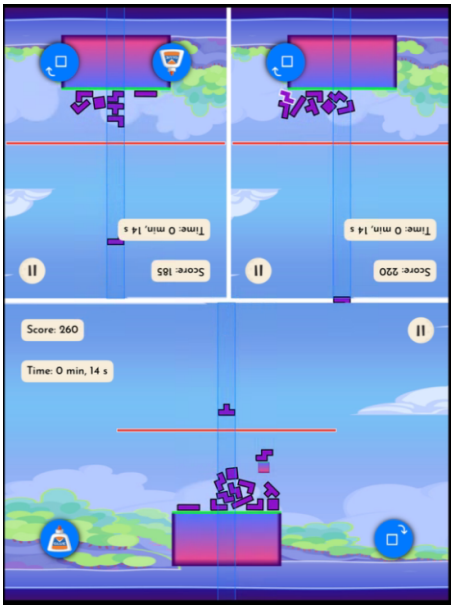
We can easily create and add more players with this design. As an example, we look into how we prototyped 3 and 4 player gameplay that we had quickly added to test what the game would be like, because of how easy it is to add due to this design.

1. Add 3 and 4 player enums to PlayersMode and Constants (2 lines addition each)
2. Add buttons for to select 3 or 4 players in PlayersSelectView
3. Create the appropriate number of game instances and position the screens properly in ContentView.
   This is actually the hardest part because of having to figure out SwiftUI UIview rotation and scaling

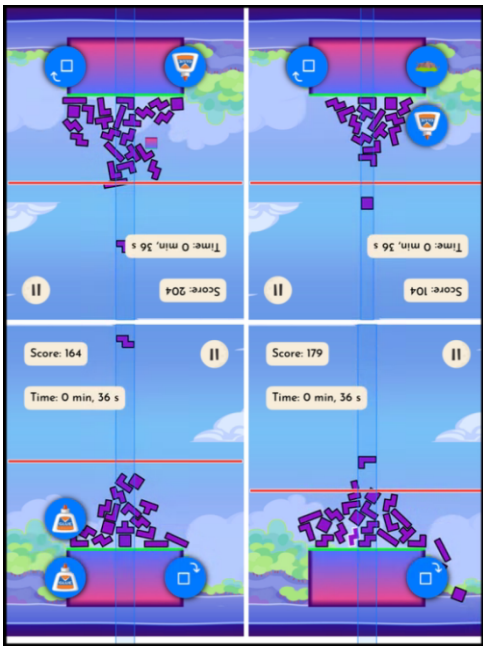
Proof of Concept Screenshots:



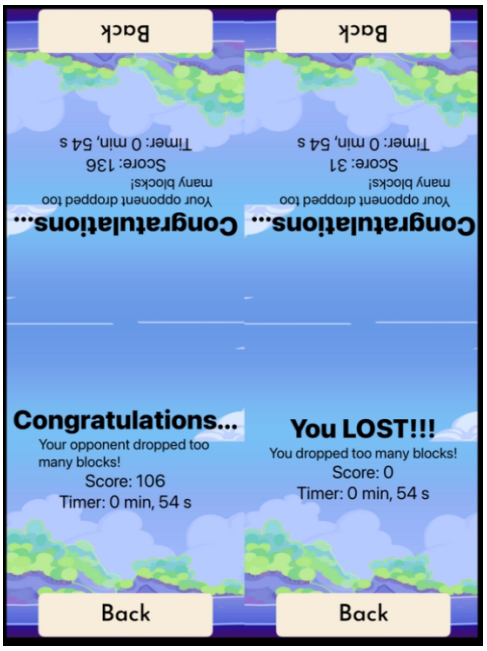
Player Select Screen for 4 Players



Gameplay Screen for 3 Players



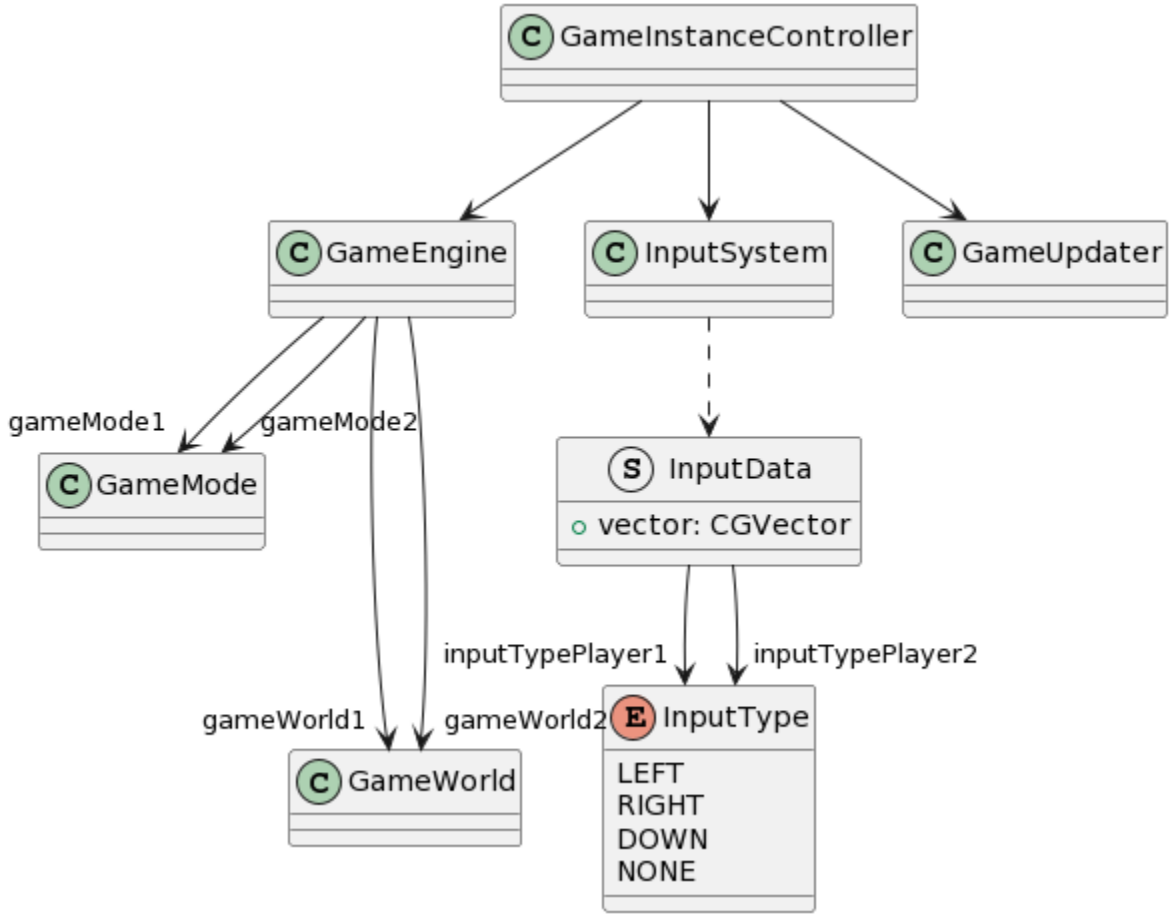
Gameplay Screen for 4 Players



Game End Screen for 4 Players

Design Considerations

ALTERNATIVE OPTION: Add multiplayer support inside GameEngine



This option may make the game run a bit better because less classes needs to be created (Only 1 GameEngine, 1 InputSystem, 1 GameUpdater).

However, the increase in performance would be very minimal as it is only a 3 classes difference. This option is also not extensible because we have to modify GameEngine whenever we want to add more players or remove players, in contrast to our current option where we just create a new instance of GameInstanceController to add a player.

InputSystem would also not be independent of player count in this case, and has to change according to the number of players as well.

Hence this option heavily violates the Open-Closed Principle.

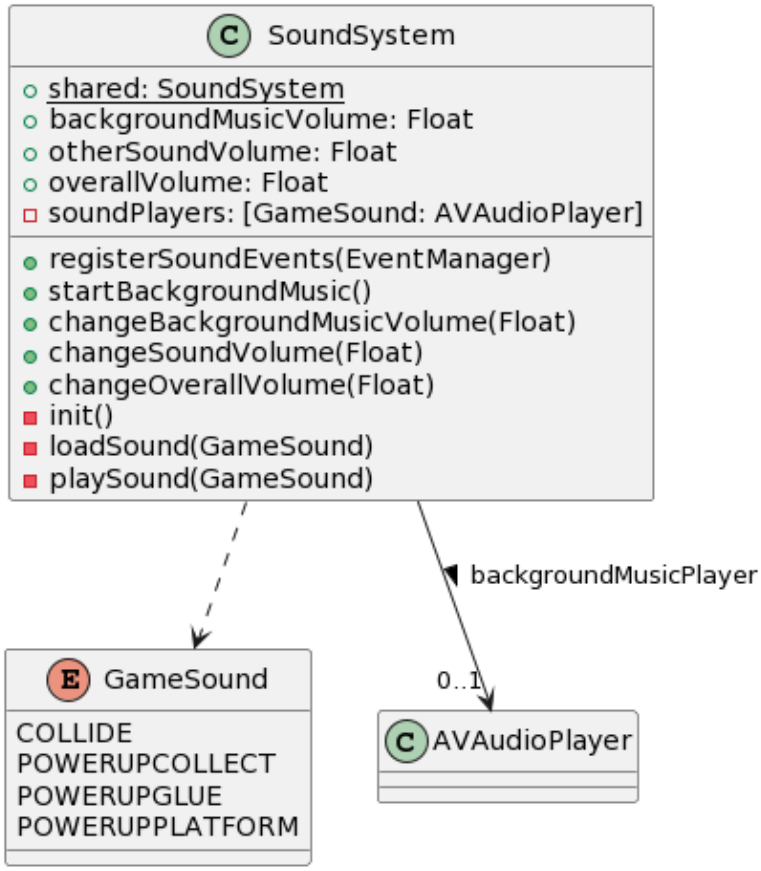
Sound

We want to have a Sound system in Tumbling Towers, in order to be able to:

- Easily play background music
- Easily play game sounds reacting to in game events
- Easily create and add new game sounds
- Easily be able to set and change volumes

This is implemented using the SoundSystem class shown below:

Module Structure



The SoundSystem uses a Singleton instance, and is an adapter for Apple's native Sound player, AVAudioPlayer. SoundSystem uses AVAudioPlayer to interface with the device's speaker in order to play the sounds (background and other sounds).

As SoundSystem is a singleton, by virtue of the design pattern the SoundSystem is instantiated automatically and the singleton instance of SoundSystem is stored in the static shared variable for easy access from other parts of TumblingTowers.

Each GameSound has a single instantiated AVAudioPlayer assigned to it (via the soundPlayers dictionary).

Runtime Structure

In the MainGameManager, when a GameInstanceController is created, the registerSoundEvents() function is called on the singleton instance. The method gives the singleton SoundSystem the EventManager currently in use, so that the SoundSystem can register certain events to play a sound when a relevant event occurs as such:

```
func registerSoundEvents(eventMgr: EventManager) {
    eventMgr.registerClosure(for: BlockPlacedEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.COLLIDE) })
    eventMgr.registerClosure(for: BlockTouchedPowerupLineEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.POWERUPCOLLECT) })
    eventMgr.registerClosure(for: GluePowerupActivatedEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.POWERUPGLUE) })
    eventMgr.registerClosure(for: PlatformPowerupActivatedEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.POWERUPPLATFORM) })
    eventMgr.registerClosure(for: GameEndedEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.GAMEEND) })
}
```

The playSound() method will find the corresponding player using the soundPlayers dictionary and use it to play the sound.

Design Considerations

Singleton or Pass around?

There are two ways which we can implement the SoundSystem.

Option 1: Creating a single instance (not singleton) and passing it around different components
In this implementation, we can create a SoundSystem instance at the app's entry point and pass it to all child components which need to call its functions, e.g. playBackgroundMusic().

Option 2: Creating a singleton instance
Alternatively, we could just create a singleton instance and allow any component across the app access to it.

Choice
We decided on **Option 2**.

When we consider the tradeoff of potential developer misuse of the singleton instance compared to additional code, the additional code needed to pass the SoundSystem around does not provide much benefit.

The SoundSystem also just plays sounds, with no other undesirable sound effects. In the worst case, if a developer misuses it, the application just sounds unusual, with no dire undesirable effects. Therefore, to reduce the complexity of the code, we chose to make SoundSystem a singleton.

Adding More Sounds

The SoundSystem takes into account OCP in its design. As such, adding a new sound is made simple. The steps required to add a new sound are:

1. Add a sound into the GameSound enum

```
enum GameSound: String, CaseIterable {
    // ... other sounds
    case NEWSOUND = "newsound.mp3"
}
```

2. Register an event for which to play NEWSOUND:

```
func registerSoundEvents(eventMgr: EventManager) {
    // ... other events
    eventMgr.registerClosure(for: NewEvent.self, closure: { [weak self] (_ event: Event) -> Void in self?.playSound(.NEWSOUND) })
}
```

3. Add the new sound, i.e. newsound.mp3 into the assets.

We see that the adding of a sound is self-contained within SoundSystem, conforming to OCP.

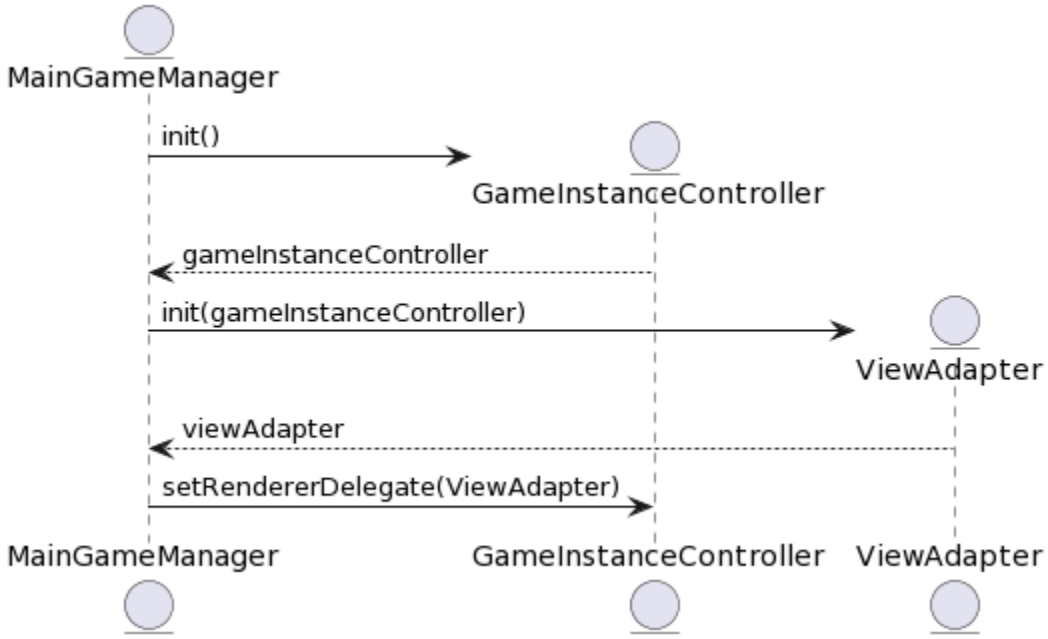
ViewAdapter

Module Structure

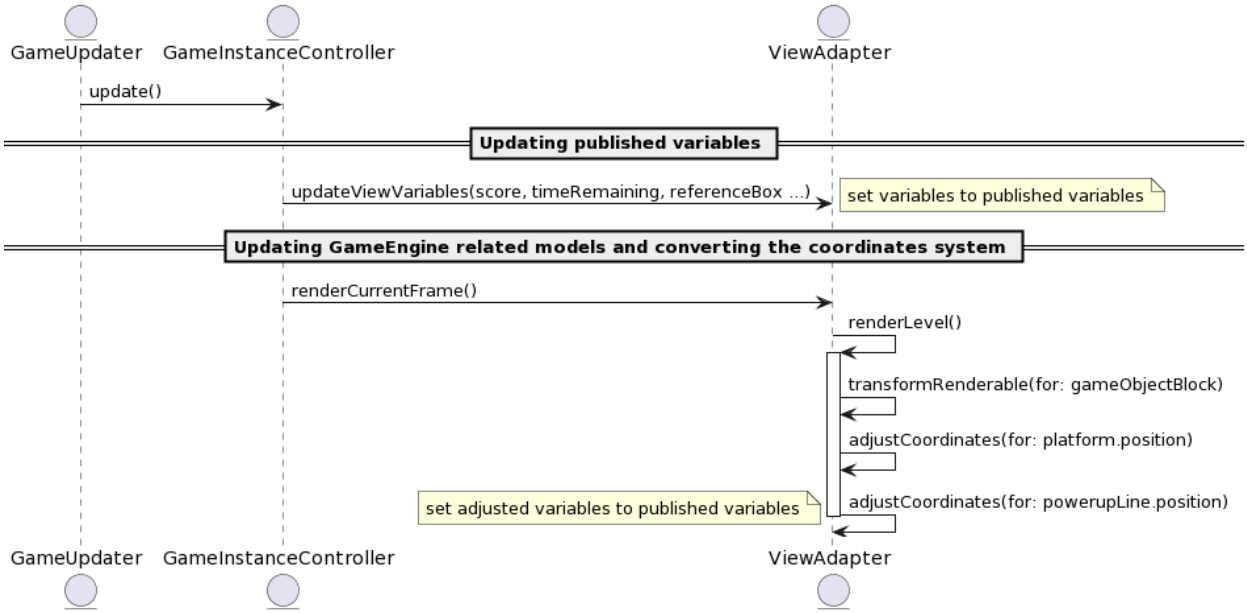
Class diagram showing ViewAdapter. ViewAdapter has a reference to GameInstanceController such that it can relay user inputs to GameInstanceController. It also conforms to the GameRendererDelegate. In the event that a new UI library is used, only ViewAdapter has to be changed as long as it conforms to GameRendererDelegate. Only relevant functions from GameInstanceController is shown for brevity.

Runtime Structure

Sequence diagram of ViewAdapter initialization.



Sequence diagram of ViewAdapter updating published variables and rendering GameInstanceController models to SwiftUI View and ViewAdapter updating the GameEngine related models and converting the coordinates system before storing in published variables



Design Considerations

When designing the ViewAdapter, we wanted it to be extensible where whenever a new UI library is used, only ViewAdapter has to be changed and GameInstanceController can be left untouched. This means there should be some dependency inversion, which is why there is the protocol GameRendererDelegate which ViewAdapter conforms to.

From here, ViewAdapter can have a reference to GameInstanceController while GameInstanceController has a function setRendererDelegate(). This makes it very easy to swap out for a new ViewAdapter implementation.

View

Module Structure

The Views are what the users would interact with. It presents the models retrieved from the ViewModels to the users, as well as receives user inputs from the user.

The hierarchy of the SwiftUI views structure is as such:

1. ContentView
    a. MainMenuView
    b. GameModeSelectView
    c. GameplayLevelView
        i. LevelView
            1. BackgroundView
            2. PowerUpLineView
                a. PathViewShape
            3. BlockView (Many)
                a. PathViewShape
            4. PlatformView (Many)
            5. GameplayGuiView
                a. PauseView
    d. AchievementsView
    e. SettingsView
    f. PlayersSelectView
    g. GameEndView (Reused across all Winning and Losing Screens)
    h. TutorialView
2. Utility Views:
    a. BackgroundView
    b. GameplayGoBackMenuView
    c. NormalGoBackMenuView

Design Considerations

Reusability

Current Implementation:
The views are designed in such a way that they are reusable, where they will render relevant instances based on what is given. Hence, the BlockView can be reused to render any of the block shapes, with the help of PathViewShape, which is an implementation of a Shape to inform the views what is the shape of the block to render. This is done to make BlockView even more extensible than our past implementation

of ViewImageManager, where now any shape of block can be rendered based on the shape of PathViewShape that we used.

Also, GameplayGuiView works in the way that when multiplayer is chosen, it will be created that number of times for each user. This proves that the views and view models integration are extremely reusable and extensible.

Past Implementation:
ViewImageManager, which is a static class that informs the views what image to render for each type of tetris block. This is done such that all image related functions and fields will be handled by ViewImageManager and if an asset is changed, only the ViewImageManager will be changed. However, for each new type of tetris block, we had to provide a new image asset for that tetris block, which we do not need to do anymore now using the PathViewShape implementation.

View Modifiers

In order to achieve a consistent style for the user interface, we make use of reusable View Modifiers to style UI components in a certain way by using some view transformations. We predefine the view transformations that we want in several View Modifiers, and apply them on all relevant text, buttons and labels in the user interface to make them all look a certain way.

This approach makes it easy to modify the style of the user interface. For example, if we want to make all the text a different color, instead of going to each Text() view construct in the code and modifying the color of each one separately, we can just modify the color transformation defined in the BodyText View Modifier and all the text in the user interface that has this view modifier applied will change color.

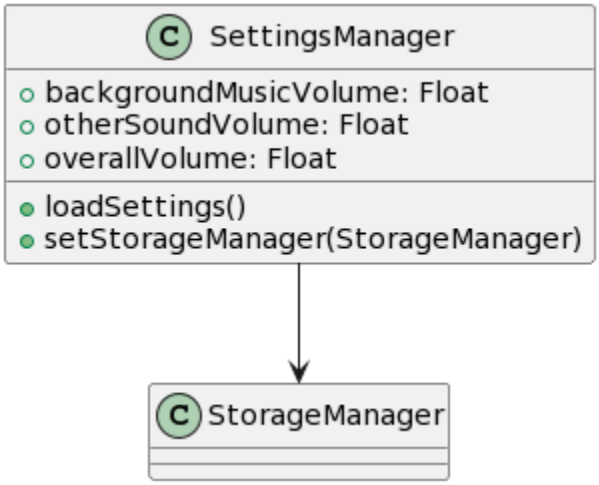
The ViewModifiers are also extremely reusable because we can apply the modifier on most SwiftUI components to instantly transform that component to match the game style that we predefined.

Current View Modifiers:

- MenuButtonText
- CategoryText
- BodyText
- CustomButton
- SecondaryButton
- PowerupButton
- GameplayGuiText
- SquareCustomButton
- GameplayBodyText
- MenuDividerLine

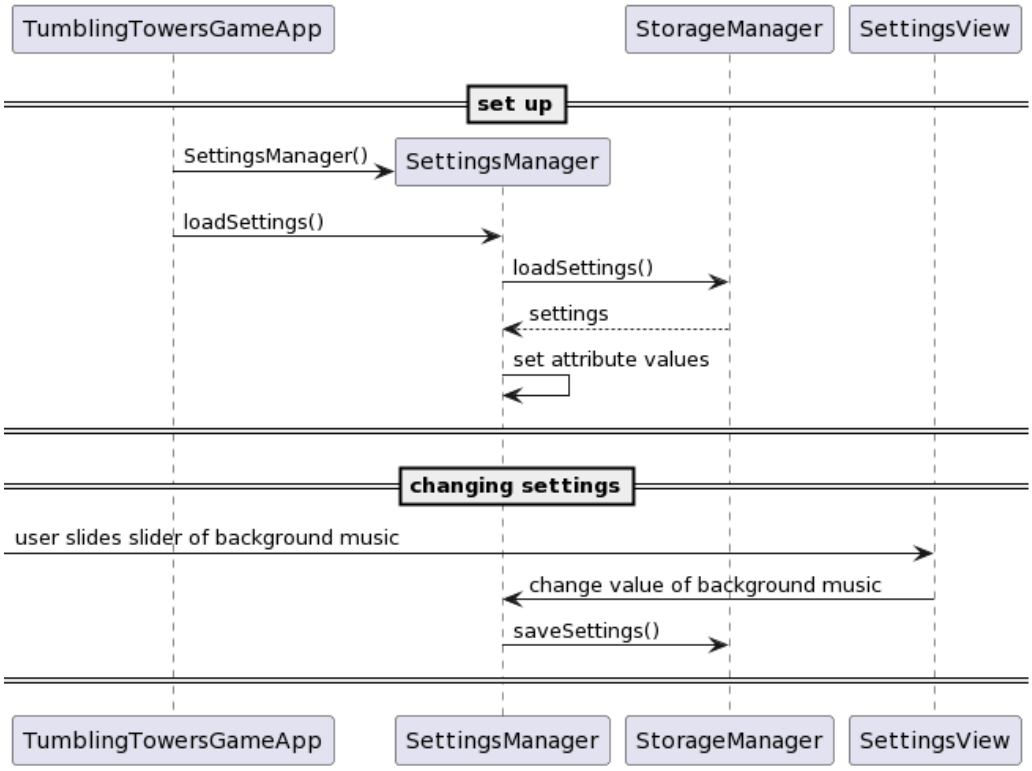
Settings

Module Structure



The SettingsManager stores the settings of the game. Currently, the only settings are the volume settings. The settings are rendered and can be changed in the SettingsView.

Runtime Structure



The top part shows the creation and set up of the SettingsManager.

The bottom section shows the sequence when a user changes the background music as an example. The sequence for the other sounds is similar to this sequence.

External libraries & third party modules

SwiftUI

SwiftUI is chosen as the choice of UI library for the application due to the speed in bringing up the views, enabling rapid prototyping through the SwiftUI previews to iterate on the MVP.

SwiftUI works in the application by having bindings (observer pattern) to the fields in the View Adapter. Compared to Storyboard, SwiftUI views can be created and reused easily. For example, in SwiftUI to create a red and bold text, a view can be created and reused just by calling the view with any text. For Storyboard, there doesn't seem to be a way to have a template to customize attributes that can be reused.

In the case where SwiftUI is to be replaced with another UI library that follows the MVVM design pattern, it can be easily swapped out and just receive updates from the view model, as well as send user input commands to the view model for processing.

NSNotificationCenter

NSNotificationCenter is used in the EventManager package.

A facade is created for NSNotificationCenter called NotificationCenterFacade and NotificationCenterFacade is used by EventManager. This ensures that if NSNotificationCenter is replaced with another library, only the functions in NotificationCenterFacade need to be replaced.

SpriteKit

SpriteKit was used in FiziksEngine. The Facade pattern was used to give a simplified interface without other features provided by SpriteKit. FiziksEngine exposes its own API and would ideally be treated like any other Physics Engine. The advantage of the Facade pattern here is that in the event that another subsystem is chosen to replace SpriteKit, only the code within FiziksEngine would need to change, the interface between the rest of the app and FiziksEngine would remain the same.

There is one caveat to this, since an SKScene needs to be presented on a SpriteView in order to begin the physics simulation, the ViewModel has to interact directly with the SKScene at this step. If SpriteKit were to be replaced with another subsystem, this would be the only part of the main app that needs to be changed.

CADisplayLink

In order to be able to synchronize the game loop with the refresh rate of the device's display, we created the GameUpdater, which makes use of SwiftUI's CADisplayLink from Apple's Core Animation framework. We chose to use CADisplayLink because it provides a very convenient way to set up, where we can easily choose which function should be repeatedly called.

The GameUpdater makes use of the facade pattern to facade out all the logic related to synchronizing the game loop with the refresh rate, handling everything from creating and managing the CADisplayLink, to making the game loop catch up on missed frames whenever there is lag. We also chose to facade out CADisplayLink into this class so that CADisplayLink is not coupled to different places in our project code and can be swapped out easily if we want to.

Core Motion

As Tumbling Towers is a game that includes gyroscopic control as a form of user to the game, we have to make use of Apple's CoreMotion to retrieve data from the device's gyroscope and accelerometer. Hence CoreMotion is used inside the GyroInput class.

GyroInput handles all the logic related to getting and processing the input from the device's gyroscope and accelerometer. We put all the code that uses CoreMotion in one class and not in any other class, so that we can easily change the implementation of the game's gyroscopic control if the way of getting such motion inputs from Apple devices changes.

AVKit and Media Player

In order to make Tumbling Towers more immersive as a game, we want to play background music as well as sounds to respond to in-game events, such as the blocks dropping or powerups activating. To do this, we have to make use of Apple's AVKit and MediaPlayer to interact with the device's speakers and volume control.

Hence, these 2 libraries are used inside SoundSystem, which handles all logic related to retrieving device volume or playing sounds through the device speakers. We put all the code that uses AVKit and MediaPlayer in one class and not in any other class, so that we can easily change the implementation of the sound system if the way of using the speakers of Apple devices changes.

Testing

Strategy

Testbed

- Tested on iOS 16 simulators: iPad 10th gen, iPad Air 5th gen, iPad Pro 11 inch 4th Gen
- Tested on hardware: iPad 8th gen (iOS 15.5)
- No sources of test data is needed

Test suites

1. GameFiziksEngine process FiziksBody
2. Updates to FiziksBody and FiziksShapeNode propagate back and forth
3. GameEngine creates FiziksBody from GameObjects, processed by GameFiziksEngine and obtains updated FiziksBody
4. GameEngineManager process Level via GameEngine and obtains updated Level
5. GameEngineManager passes inputs to GameEngine
6. InputSystem determines inputs from users through screen tapping or gyroscope of device
7. ViewImageManager has information on image assets
8. Integration of views and GameEngineManager
9. EventManager able to post events and subscribe closures to events
10. Storage able to store and load data on the device
11. ViewAdapater stores published variables and changes the coordinates system of GameEngine related models
12. Achievements shown to user and persists across sessions
13. PowerUp can be generated and fired during the game
14. Various GameModes should be able to determine different winning and losing conditions of the game

More test suites and detailed test cases can be found in the appendix.

Main strategy for testing is the top down approach as the likelihood of defects on the top modules is higher and has a higher impact. For example, mocks are created to replace other referenced classes during testing as shown in the usage of MockFiziksBody, MockFiziksSceneUpdateDelegate and MockFiziksContactDelegate during the testing of GameFiziksEngine and FiziksContact. GameFiziksEngine in this case integrates the mocks and tests the important functionality of processing FiziksBody.

Models and view models are tested using the glassbox approach, as all functionalities of the code should perform as desired and bugs can be pinpointed. Integration testing that involves views will be the black box approach and more towards acceptance testing, mainly to test the behavior of the application from a user point of view.

Expected classes of errors found from the unit testing would be compile time checks. We are aware that more run time assertions should be added to ensure the data types representations, inputs and outputs are as designed.

Models and view models will be tested via unit testing and integration testing. The ease of testing will also be beneficial in determining how coupled the models are and whether the models are really in a proper facade. Integration testing of views with view models will be done via manual user testing, as described in the appendix.

Reflection

Evaluation

Sprint 3

We split our sprint in a more structured manner, splitting into Sprint 3.1 and Sprint 3.2. Sprint 3.1 was mostly focused on adding any last features and refactoring (from Sprint 2's feedback), while Sprint 3.2 was focused on bug fixing, code cleanup, and documentation. We implemented a feature freeze deadline (before STePS) to ensure that we would have sufficient time to fix any bugs in our new features. We realized that this was crucial to be safe and to ensure that our app would be working during the project showcase.

Overall, we feel that this project was indeed a good hands-on experience in the software engineering process. Throughout the project, we engaged in lengthy discussions regarding various ways to solve problems, and weighing the pros and cons so that we could choose the most suitable solutions. We also refactored our application as we continued adding new features, to ensure that no component is too big which ensures SRP and SoC.

Sprint 2

We allocated the tasks in Sprint 2 in a way such that we would interact/work with parts of the codebase that we did not interact with much in Sprint 1. This serves as a way for us to check the maintainability of our code and evaluate the ease of adding new features. This allows us to determine the quality of our code, based on how easy it is for "new" developers to add features. This also allowed us to identify any design issues or points of improvement for our code.

Sprint 1

With regards to workload allocation, we managed to allocate work well, allowing us to meet our deadline and complete the tasks we set out to do within this sprint. Nonetheless, there are still certain ways we can improve our processes. For example, we could better decide and specify APIs beforehand.

While we did try to decide what APIs would be needed, for example in FiziksEngine, it is sometimes difficult to know all the APIs which are needed before development begins. Perhaps we could incorporate better planning to allow us to better decide what APIs to expose in components.

Lessons

Sprint 3

In this sprint, we learnt how to ensure that our components would conform to OCP, and a lot of effort was placed in refactoring our code as such. We also learnt that the Adapter pattern is useful in ensuring that it would be easy to swap out third party libraries such as SwiftUI for the View.

We learnt that one of the most useful ways to improve OCP is to use the enum-dictionary combination, since this removes the need to modify existing classes to extend if-else and switch-case clauses when new cases are added.

Sprint 2

We learnt to make our pull requests more atomic, in order to make the merging process less time-consuming, allowing us to focus on more important aspects of the project. This allowed us to have an easier time working in parallel, as each pull request was not too bloated, making it easier for us to merge others code into our own branches.

Sprint 1

We learnt that the Facade pattern is very useful to help decouple our project from the other 3rd party libraries that we are using. Currently we are using it to facade out external libraries like SpriteKit, CADisplaylink, CoreMotion, NSNotificationCenter, AVKit and MediaPlayer. Although it is difficult to design and create the facade, it would benefit us in the long run in case any of the libraries have to be switched out because they have become depreciated, or if there are better options in the future.

Known bugs and limitations

1. One limitation is the fact that our FiziksEngine requires a blank SpriteView to render on. This means that the model has a dependency on the view, which is not ideal. However, this is not avoidable when using SpriteKit. In the event that SpriteKit were to be replaced with another Physics Engine, this is the only aspect that requires change. This is further elaborated on in the FiziksEngine section.

2. We are using a random number generator for the creation of blocks and powerups. Since the swift `random(in:)` method does not allow seeding, we opted to use `drand48()` instead. This is a legacy method and is flagged by swiftlint but we consider this a justifiable reason to violate

swiftlint warnings. However, there is a bug where the random number generator is still not seeded properly. With the same seed passed into two instances of the random number generator, we expect both RNGs to produce the same sequence of numbers but the result is different.

Appendix

Glossary

Term	Meaning
CMB	Currently moving block
SoC	Separation of concerns
OCP	Open Closed Principle
SRP	Single Responsibility Principle

Test Cases

Integration tests

1. Orientation
    a. Application should only be in portrait mode and not landscape mode
    b. Orientation of each individual games (during multiplayer mode) should appear user friendly and intuitive
2. Devices
    a. Application should display the game in devices / simulators as stated in the testing strategy section.
    b. Application should be scaled to device screen
    c. Application should work on devices with >= iOS 15.5
3. Menus
    a. Main Menu
        i. When the application is first started, the main menu screen should be shown.
        ii. When “Play” is selected, the user should proceed to the Player Selection Screen
        iii. When “Achievements” is selected, the user should proceed to the Achievements Screen
        iv. When “Settings” is selected, the user should proceed to the Settings screen.
        v. When “?” is selected, the user should proceed to the Tutorial screen.
        vi. If none of the 4 options are selected, the user should not proceed anywhere.
    b. Player Selection Screen
        i. When “Singleplayer” is selected:
            1. Gameplay Screen should only show a single screen (one set of platform, score UI, pause button, etc.) regardless of game mode
            2. User should be able to use Gyro or Tap input (depending on the setting in Settings)
        ii. When “Multiplayer” is selected:
            1. Gameplay Screen should show two screens regardless of game mode, where the device is split horizontally.
            2. Users should only be able to use Tap input (regardless of the setting in Settings)
        iii. When “Back” is selected, the user should head back to the Main Menu.
    c. Game Mode Selection Screen
        i. When “Survival ” is selected, the user should start up the game with the Survival Game Mode.
        ii. When “Sandbox” is selected, the user should start up the game with the Sandbox Game Mode.
        iii. When “Race the Clock” is selected, the user should start up the game with the Race the Clock Game Mode.

iv. When “Scale the Heights” is selected, the user should start up the game with the Scale the Heights Game Mode.
v. When “Back” is selected, the user should head back to the Player Select Screen.
vi. If none of the options are selected, the user should not proceed anywhere.

d. Achievements Screen
    i. By default, there should be 4 achievements shown: BobTheBuilderI, BobTheBuilderII, SkyscraperI and SkyscraperII.
        1. Each Achievement should have its own description of how it can be achieved, along with the player’s progress in achieving it.
        2. There should also be a ‘tick’ or ‘cross’ depending on whether or not the achievement has been achieved.
    ii. When “Back” is selected, the user should head back to the Main Menu.
    iii. If none of the options are selected, the user should not proceed anywhere.

e. Settings Screen
    i. Volume
        1. When the Overall Volume Slider is moved, the device’s system volume should change according to the position of the slider.
        2. When the Background Music Volume Slider is moved, the background music volume should change according to the position of the slider.
        3. When the Other Sounds Volume Slider is moved, the volume of other game sounds should change according to the position of the slider.
        4. When none of the sliders are moved, the volumes of the various sounds should not change.
    ii. Input Options
        1. The input option should be able to be set as Tap or Gyro. This should set the input system for singleplayer that is used when the game starts.
        2. The input option should not affect the input system used for multiplayer where it should stay as Tap input.
    iii. When “Reset Achievements” is selected in the Settings Screen, all achievements should be cleared to their default state (not achieved and progress is 0).
    iv. When “Back” is selected, the user should head back to the Main Menu.
    v. If none of the options are selected, the user should not proceed anywhere.

f. Game End Screen (Win and Lose Screens)
    i. An appropriate win or lose screen should be shown based on the selected game mode and the winning or losing condition that was triggered.
    ii. The current score and current time should also be shown to the user.
    iii. When “BACK” is selected, the user should head back to the Main Menu.
    iv. If none of the options are selected, the user should not proceed anywhere.

4. Input System
    a. When using Drag Input
        i. When the user drags their finger towards the left

            1. the currently moving block should be moving left proportional to the amount that the user dragged their finger left
            2. the currently moving block should not go beyond a certain invisible level boundary
            3. the currently moving block should not speed up its descent if user drags down their finger
        ii. When the user drags their finger towards the left,
            1. the currently moving block should be moving right proportional to the amount that the user dragged their finger right
            2. the currently moving block should not go beyond a certain invisible level boundary
            3. the currently moving block should not speed up its descent if user drags down their finger
        iii. When the user drags their finger down,
            1. the currently moving block should be moving down at a constant, but faster rate than the normal falling speed
            2. the currently moving block should not move left/right if user drags their finger left/right
    b. When using Gyroscopic Input
        i. When the user plays using a hardware device and the gyroscope input system is used, and the device is tilted to the left,
            1. the currently moving block should be moving left proportional to the amount that the user tilted their device left
            2. the currently moving block should not go beyond a certain invisible level boundary
            3. the currently moving block **should** speed up its descent and stop moving left if user drags down their finger
        ii. When the user plays using a hardware device and the gyroscope input system is used, and the device is tilted to the right,
            1. the currently moving block should be moving right proportional to the amount that the user tilted their device right
            2. the currently moving block should not go beyond a certain invisible level boundary
            3. the currently moving block **should** speed up its descent and stop moving right if user drags down their finger
        iii. When the user drags their finger down,
            1. the currently moving block should be moving down at a constant, but faster rate than the normal falling speed
5. Blocks
    a. New blocks should appear at the top of the screen, on top of the platform and descend automatically at a constant rate.

    b. Blocks should stop moving when it hits a platform or other blocks, and should be subjected to game physics such as gravity (blocks should fall off if it is off balanced)
    c. Blocks that do not fall off should remain there unless hit by other blocks
    d. After the block currently being controlled collides with another block, the user should lose control of the currently falling block. A new block should be inserted at the top center of the screen and the user should gain control of that block.
    e. When tapping the rotate button
        i. The block will rotate clockwise by 90 degrees
        ii. If the block is in a position such that rotating it will collide with something, it will still rotate and the new position will be after the collision is resolved
6. Platforms
    a. Platforms should remain at the same place with the same dimensions
    b. The main platform should start at the middle bottom of the screen
7. Game Modes
    a. Survival
        i. The winning condition should be triggered when there are 30 blocks placed onto the platform successfully at any one time.
        ii. If a block falls off the screen, it should not be counted towards the winning condition even if it has successfully been placed onto the platform before.
        iii. The losing condition should be triggered when 3 or more blocks have dropped off the platform.
        iv. The winning condition should not be triggered when there are less than 30 blocks placed onto the platform at any one time.
        v. The losing condition should not be triggered when less than 3 blocks have dropped off the platform.
        vi. The score should decay over time.
        vii. If the game is completed within 90 seconds, the user should receive a pro-rated bonus score.
    b. Sandbox
        i. There should be no winning or losing condition.
        ii. The game will run indefinitely until the application is shut down.
    c. Race the Clock
        i. The time limit for singleplayer should be 60 seconds, time limit for multiplayer is 30 seconds.
        ii. The winning condition should be triggered when there are 30 blocks placed onto the platform successfully at any one time within the time limit.
        iii. If a block falls off the screen, it should not be counted towards the winning condition even if it has successfully been placed onto the platform before.
        iv. The losing condition should be triggered when the user runs out of time.
        v. The winning condition should not be triggered when there are less than 30 blocks placed onto the platform at any one time.
        vi. The losing condition should not be triggered when the timer has not run out yet.

        vii. The score should decay over time.
    d. Scale the Heights
        i. The winning condition should be triggered when the 10th power up line is reached for singleplayer, 5th power up line for multiplayer .
        ii. The losing condition should be triggered when 20 or more blocks have dropped off the platform.
        iii. The winning condition should not be triggered when the 10th power up line is not reached yet for singleplayer, 5th power up line not reached yet for multiplayer .
        iv. The losing condition should not be triggered when less than 20 blocks have dropped off the platform.
        v. The score should decay over time.
        vi. If the game is completed within 60 seconds, the user should receive a pro-rated bonus score.
8. Power Ups
    a. When block intersects the powerup line & is stationary
        i. Power up line should move upwards
        ii. If user had no power ups prior:
            1. A power up button should appear
        iii. If user had power ups prior:
            1. A **new** power up button should appear
    b. When tapping any power up button at position i
        i. The ith button should disappear
        ii. The other buttons should not move & should remain
    c. When tapping glue power up button
        i. Falling block should have a white outline
        ii. When it contacts a surface (platform/block), it should stick to that surface
        iii. Subsequent blocks which touch the block with the white outline should **not** stick to it
    d. When tapping platform power up button
        i. Platform should be created within the x coordinate range of the platform
        ii. Platform should be under the next power up line
        iii. Blocks should be able to be placed on the platform
        iv. Future glue blocks should be able to stick to the platform
        v. Platform should not intersect any other objects when being inserted
    e. Power up sequence should be deterministic based on the seed
        i. If user plays game with same seed twice, power up sequence received will be the same
9. Sound System
    a. When the application is started, background music should start playing on loop.
    b. When a block collides with another block in game, the block collide sound should play.
    c. When the powerup line is reached, the powerup line reached sound should play.

    d. When glue powerup is activated, the glue powerup sound should play.
    e. When platform powerup is activated, the platform powerup sound should play.
10. Achievement System
    a. Progress for achievements should be updated only if the user is in singleplayer.
    b. Progress for achievements should not be updated if the user is in multiplayer.
11. Event Manager
    a. When closures from multiple packages is registered for an event and the event is fired, all the closures should be run
    b. There should only be 1 eventManager per game that is passed to relevant classes that uses it
    c. Should allow classes to register closures for a certain event
    d. Should allow classes to post events
    e. EventManager should work with other packages such as Achievements, GameMode, PowerUp and Sound.
12. Storage
    a. When the application starts, previous settings such as the SoundSystem volumes and the previous Achievements should be loaded and updated
    b. When the settings are updated from user gestures, it should automatically be saved
    c. When a new Achievement is obtained, it should be added to the existing list of achievements and saved
    d. Stats should be internally saved and loaded on every setup. The updated stats should be persisted as well
    e. There should only be 1 StorageManager that is passed around to classes that use it
    f. The storage system should be easily swapped out if another storage method such as CoreData or SQL is used
13. ViewAdapter
    a. All views should only have a reference to ViewAdapter instead of direct access to GameInstanceController
    b. ViewAdapter should do the necessary conversion (coordinates) from the GameEngine models to the coordinates that is used by the SwiftUI views
    c. ViewAdapter should periodically update its published variables from the GameEngine models
    d. ViewAdapter should relay the user inputs to the GameInstanceController
    e. ViewAdapter should be deinit when it is not used anymore by the game

Detailed Task Schedule + Task List

Task	Allocation	Deadline
Sprint 1		
Learn UI Libraries	All Team Members	13 March
Learn SpriteKit	All Team Members	13 March
Set up Physics Engine	Taufiq	13 March
Facade SpriteKit	Taufiq Teng Foong	13 March
Set up Game Engine	Teng Foong	13 March
Set Up Views	Yong Ler	13 March
Set Up ViewModels	Elvis Yong Ler	13 March
Implement Input System	Elvis	13 March
Merge Components and packages	Yong Ler	15 March
GameUpdater	Elvis	17 March
Gyroscopic Control	Elvis	17 March
Plan powerup framework	Taufiq	17 March
Plan Gamemode framework	Yong Ler	19 March
Integrate input system with game engine	Elvis	19 March
Complete Sprint Report	All Team Members	19 March
Sprint 2		
Event System	Yong Ler	25 March (Saturday)
Powerups (requires event system)	Taufiq	28 March (Wednesday)
Achievements System	Teng Foong	31 March (Friday)
Stats Tracking System	Teng Foong	31 March (Friday)

Game Mode (requires event system) - Game Mode 1 - Game Mode 2	Elvis	31 March (Friday)
Sound (requires event system)	Elvis	31 March (Friday)
Main Menu, Settings Menu, Win and Lose Screens, View Modifiers	Elvis	31 March (Friday)
Storage Manager - JSON - Settings	Yong Ler	28 March (Wednesday)
Couple SKNode and FiziksBody	Teng Foong	25 March
Sprite Kit Renders Shape CGPath instead of Image	Taufiq	25 March
Platform - Create the actual body - Render it properly - Boundary should be at the platform not the screen side - include some padding at the side - If block falls off platform, post an event - Power up line	Taufiq	28 March (Wednesday)
Input system update - Drag instead of tap - Swipe down to slam - Gyro try flicking down if cannot also swipe down to slam - Add power up button	Taufiq	31 March (Friday)
Combining bodies	Taufiq Teng Foong	31 March (Friday)
Complete Sprint Report	All Team Members	2 April (Sunday)
Sprint 3.1		
Multiplayer mode	Taufiq	7 April (Friday)

	Elvis	
Rendering of achievements and storage	Yong Ler Teng Foong	8 April (Saturday)
Gameplay GUI (Score System + Timing + Blocks Placed/Blocks Dropped)	Elvis	
Pause Menu	Taufiq	5 April (Wednesday)
Image Assets for Game and Rendering	Taufiq	5 April (Wednesday)
Race Against Time Gamemode	Elvis	8 April (Saturday)
View Adapter	Yong Ler Elvis	7 April (Friday)
FEATURE FREEZE (for STEPS) - 9 April 2359		
Sprint 3.2		
Code Cleanup	Teng Foong Elvis	16 April (Sunday)
Refactoring Code - Game Engine God class	Taufiq	13 April (Thursday)
Fix any existing design problems	Teng Foong	15 April (Saturday)
Documentation and More Diagrams	All	16 April (Sunday)

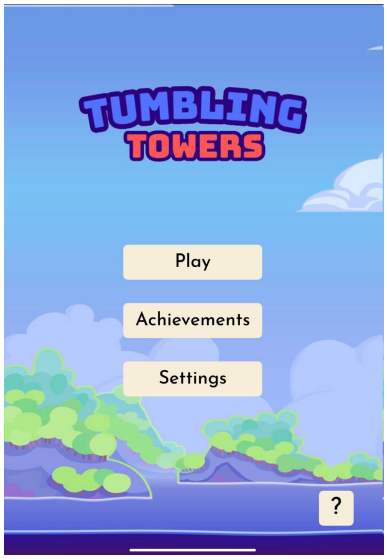
Individual contribution and roles

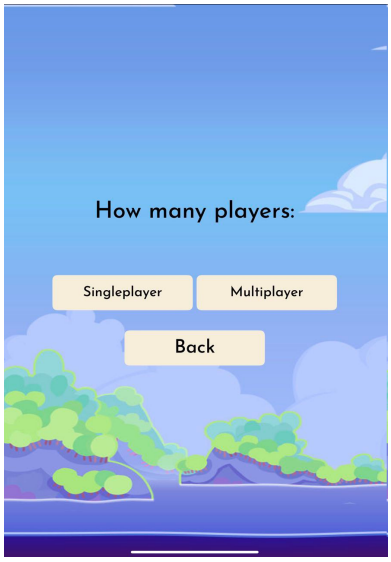
There are no external partners involved in this project.

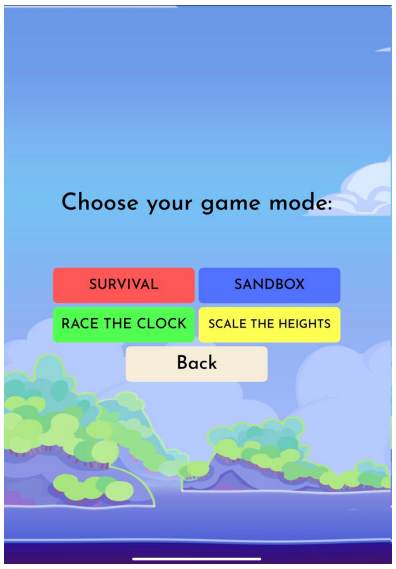
Member	Contributions
Yong Ler	- SwiftUI views - SwiftUI view models - Integration of packages and refactoring - EventManager - Storage - ViewAdapter
Elvis	- SwiftUI views - SwiftUI view models - Input System - Game Modes - Sound - Settings - Add support for Multiplayer in Views and GameInstanceController - Score system & UI - Navigation between Views - Game Menus - Fix Memory Leak
Teng Foong	- Tetris Shape Randomizer - Game Engine - Fiziks package (SpriteKit facade) - Improvements in facade for SKNode - Achievements System (and views) - Stats Tracking System - Game World Level
Taufiq	- Fiziks package (SpriteKit facade) - Power up system - Rendering Tetris shape blocks - Rotation & power up buttons - Power up line - Level boundaries - Update Input System - Refactor GameEngine - Refactor MainGameManager, GameInstanceController & Views to support Multiplayer

GUI Sketches/Screenshots

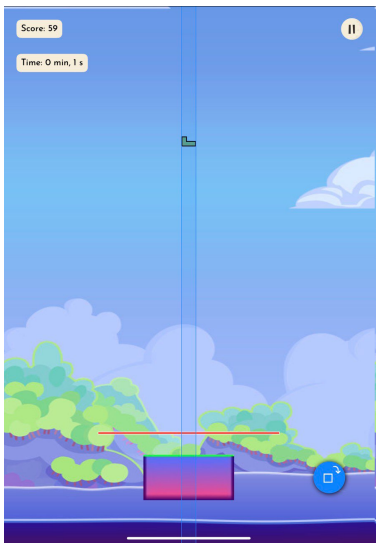
Menu Screens

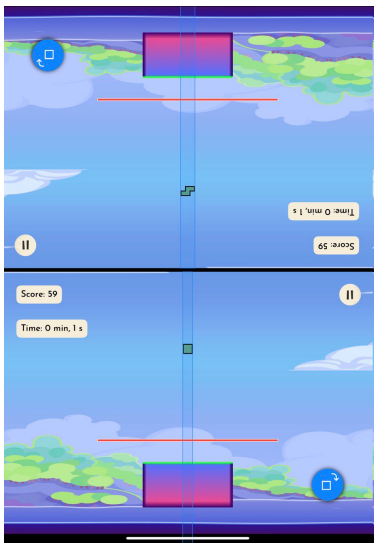




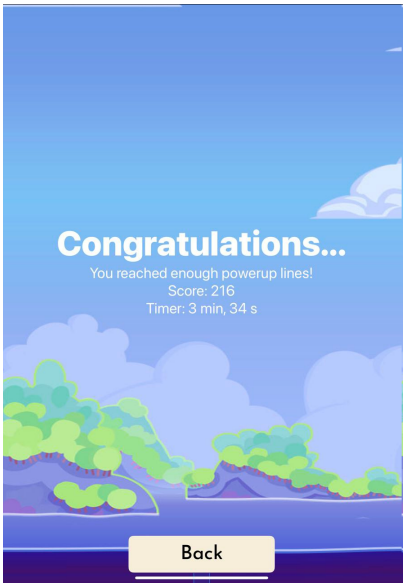


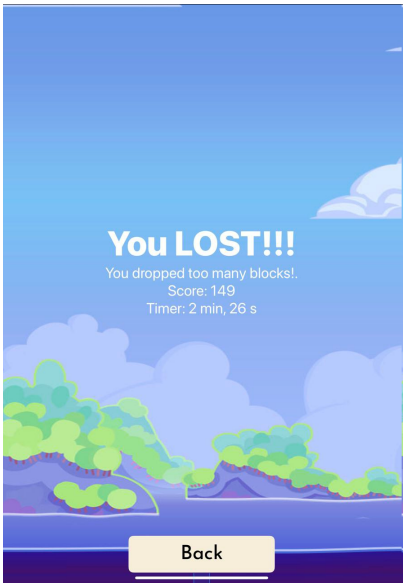
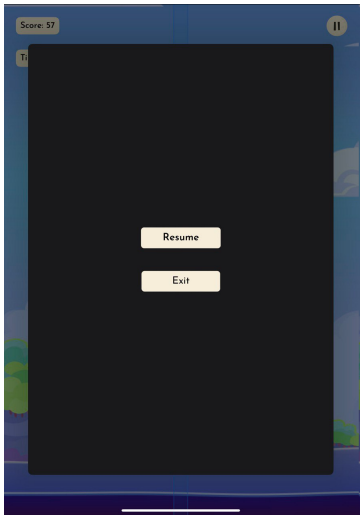
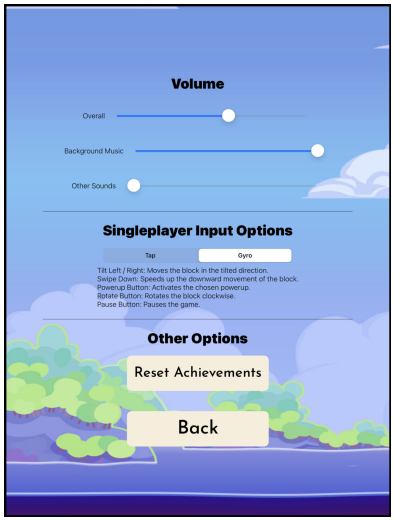
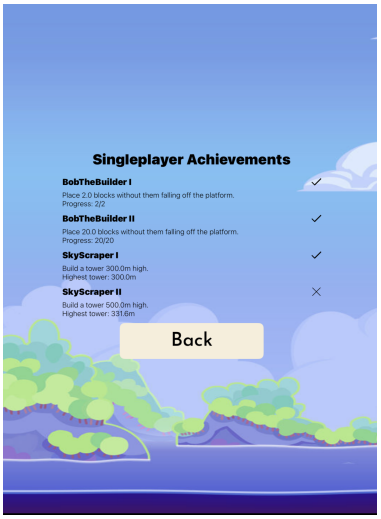
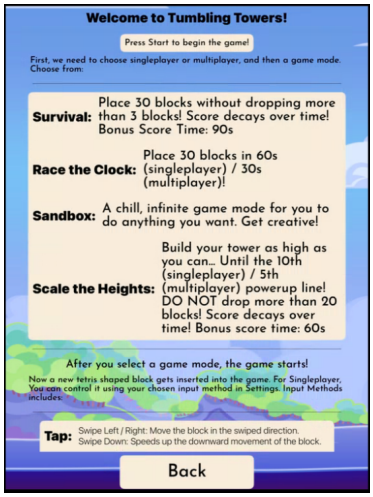
Game Screens





Game End


Congratulations...
You reached enough powerup lines!
Score: 216
Timer: 3 min, 34 s
Back


You LOST!!!
You dropped too many blocks!.
Score: 149
Timer: 2 min, 26 s
Back

Other Screens